TU WIEN Informatics

Foundations for the Security Analysis of Distributed Blockchain Applications

PhD THESIS

submitted in partial fulfillment of the requirements for the degree of

Doctor of Technical Sciences

within the

Vienna PhD School of Informatics

by

Clara Schneidewind, BSc
Registration Number 01652950

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Matteo Maffei

External reviewers:
Arthur Gervais. Imperial College London, United Kingdom.
Bernhard Scholz. The University of Sydney, Australia.

Vienna, 8th March, 2021

Clara Schneidewind	Matteo Maffei

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Declaration of Authorship

Clara Schneidewind, BSc

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 8th March, 2021

Clara Schneidewind

Acknowledgements

I want to sincerely thank my advisor Matteo Maffei for guiding me through this Ph.D. Over and above the valuable academic guidance, his optimism and enthusiasm for research always was a source of motivation and cheerfulness to me and helped me overcome the desperate phases of my Ph.D. life. On top of that, I am deeply grateful that he never stopped trusting and promoting me. Having him as my 'PR person' was the luckiest find of my Ph.D.

I would also like to thank my collaborators, and foremostly my group in Saarbrücken and then Vienna. Starting from my first day in the group, I was welcomed with that amazing mix of warmth and humor that withstood the relocation to Vienna and all changes in the group. Thank you all for the fun lunches, parties, and, of course, cake o'clock gatherings. Despite thanking everyone for being so incredibly funny, helpful, and good-hearted, I would like to put on record that this group comes with an extraordinarily high average level of baking and cooking skills. I cannot fully extend this compliment to the dancing skills, but I would still like to wholeheartedly thank my beloved 'Swing Coast West Whatever' dancing crew: Mauro, Pedro, and Barbara. Similarly funny (but in equal measure also devastating) experiences I was lucky to share with Ilya and Markus on our crazy *eThor* journey where we experienced the spirit of *HoRSt*. Big thanks also to Niklas for suffering from and with me through that Ph.D., and for – depending on what the situation required – being either a bastion of calm or showing off with his balloon animal tying skills for cheering me up. Further, I would like to thank Pedro, without whom the quality of my life in Vienna would have tremendously dropped. Thank you for all the breakfasts that prepared me for exhausting days at work and the dinners that saved me afterward. On top of that, I cannot thank him enough for all the support with my application and thesis and for always being such a great friend. Also, I am more than grateful to Francisco for accompanying me on my next steps and, in this way, laying the perfect foundation for bringing our group's spirit to Bochum.

I also want to thank Prof. Smolka for being my mentor over all these years. The confidence that I gained from his experienced and considered advice helped me through difficult times during my Ph.D., and thanks to him, the 5th floor of E1.3 always stayed my safe shelter.

On top of all the support from my professional environment, I was incredibly lucky to have always been backed by great friends and my loving and inspiring family. I am particularly grateful to my friends in Saarbrücken, who made the city a home for me. A ‚fettes Merci' to Noemi, Norine, Yannick, Chris, Jana, Kathrin, Fabian, Caro, and all the others. Special thanks go to Sebastian, who always kept being there for me despite all the burdens that this Ph.D. brought to both of us. Finally, I would like to thank Sophia, Jacob, my parents, and my grandparents. I consider myself

highly privileged to have a family of strong and independent individuals who can challenge and push you when needed, and that is still the place that I would always return to in the midst of a pandemic.

Kurzfassung

Kryptowährungen ermöglichen nicht nur Geldtransfers in Abwesenheit eines vertrauenswürdigen Dritten, sondern auch die Ausführung verteilter Anwendungen. Aufgrund der rasanten Entwicklung von Kryptowährungen wurden die theoretischen Grundlagen solcher Anwendungen bisher nicht gründlich untersucht. Dies ist besonders problematisch, da diese Anwendungen Geldflüsse kontrollieren und Sicherheitslücken regelmäßig schwere finanzielle Verluste verursachen.

In dieser Arbeit stellen wir zwei systematische Ansätze zur zuverlässigen und therotisch fundierten Verifikation verteilter Blockchain-Anwendungen vor. Dazu betrachten wir die Kryptowährungen mit der höchsten Marktkapitalisierung, Bitcoin und Ethereum. In Ethereum werden verteilte Anwendungen als *Smart Contracts* realisiert. Das sind reaktive Programme, die in Ethereums mächtiger Skriptsprache geschrieben sind. Bitcoin hingegen unterstützt nur eine simple Skriptsprache, und komplizierte Anwendungen werden als kryptographische Protokolle realisiert, die die gewünschte Funktionalität letztendlich auf die Ausführung mehrerer einfacher Smart Contracts zurückführen. Daher liegt die Herausforderung bei der Verifizierung verteilter Anwendungen in Ethereum in der Charakterisierung und Abstraktion der Semantik von Ethereums Skriptsprache, während dies in Bitcoin eine systematische Analyse kryptographischer Protokolle erfordert.

In dieser Dissertation formalisieren wir zunächst die vormals unterspezifizierte Semantik von Ethereums Skriptsprache EVM-Bytecode und implementieren die Semantik in dem Beweisassistenten F*. In diesem Zusammenhang charakterisieren wir relevante generische Sicherheitseigenschaften von Smart Contracts, welche reale Angriffsszenarien ausschließen.

Anschließend geben wir einen Überblick über automatisierte statische Analysetools für Ethereum Smart Contracts, und diskutieren Schwachstellen in den semantischen Grundlagen dieser Tools, sowie deren praktische Auswirkungen auf die Analyseergebnisse. Davon motiviert präsentieren wir unser eigenes Analysetool, welches beweisbare Sicherheitsgarantien liefert und gleichzeitig eine konkurrenzfähige Performance zeigt. In diesem Zuge entwickeln wir auch ein allgemeines Framework für die modulare Entwicklung automatischer statischer Analysetools.

Abschließend untersuchen wir die Sicherheit von Payment Channel Networks für Bitcoin. Payment Channel Networks sind Protokolle für effiziente und kostengünstige Zahlungen zwischen Bitcoin-Nutzern und damit ein vielversprechender Lösungsansatz für die Skalierbarkeitsprobleme von Bitcoin. Wir beschreiben ein Sicherheitsproblem der bestehenden Implementierung dieser Protokolle in Bitcoin und geben eine formale Charakterisierung relevanter Sicherheits- und Anonymitätskonzepte. Zum Abschluss entwickeln wir ein kryptographisches Primitiv für die Konstruktion von Payment Channel Networks mit formalen Sicherheitsgarantien.

Abstract

Cryptocurrencies do not only allow for money transfers in the absence of a trusted third party but also enable the execution of distributed applications. Due to the rapid pace of development of cryptocurrencies, the foundations of such applications have not been rigorously studied. This is particularly problematic since in these applications, real money is at stake, and security breaches regularly cause severe financial losses.

In this thesis, we present two systematic approaches to reliably verify the security of distributed blockchain applications based on formal foundations. To this end, we focus on the cryptocurrencies with the highest market capitalization, Bitcoin and Ethereum. In Ethereum, distributed applications are realized as *smart contracts*, reactive programs written in Ethereum's expressive scripting language. In contrast, Bitcoin supports only a basic scripting language, and advanced applications are realized as peer-to-peer cryptographic protocols that resort to the execution of simple smart contracts in case of disputes among peers. As a result, the challenge in verifying distributed applications on the Ethereum blockchain lies in the study and abstraction of the semantics of Ethereum's evolved scripting language, whereas Bitcoin, the study of distributed applications, requires a systematic analysis of the cryptographic protocols.

In the thesis, we first formalize the formerly under-specified semantics of Ethereum's native smart contract language EVM bytecode and implement the semantics in the proof assistant F*. In this context, we formally characterize relevant generic properties for smart contract security, which capture real-world attack scenarios.

We then survey existing automated static analyzers for Ethereum smart contracts unveiling the weaknesses in the semantic foundations of these tools and the practical impact of these weaknesses on the analysis results. Based on these findings, we propose our own automatic static analysis tool for Ethereum smart contracts, which comes with a rigorous soundness proof while still showing competitive performance. In this course, we also propose a general framework for the modular and semantic-driven development of automatic static analyzers.

Finally, we study the security of payment channel networks for Bitcoin. Payment channel networks are distributed protocols that allow for efficient and cheap payments between Bitcoin users and offer a promising solution to Bitcoin's scalability problems. We unveil a security issue in Bitcoin's existing payment channel network implementation and formally characterize the relevant security and privacy notions in this context. We further develop a cryptographic primitive for the construction of payment channel networks with formal security guarantees.

List of Publications

[GMS18b] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *International Conference on Principles of Security and Trust - 7th International Conference, POST 2018*, pages 243-269. Springer, 2018.

[GMS18a] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. Foundations and Tools for the Static Analysis of Ethereum Smart Contracts. In *30th International Conference on Computer Aided Verification, CAV 2018*, pages 51-78. Springer, 2018.

[MMSS+19] Giulio Malavolta, Pedro Moreno Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability. In *The Network and Distributed Systems Symposium, NDSS 2019*. 2019.

[SSM20] Clara Schneidewind, Markus Scherer, and Matteo Maffei. The Good, The Bad and The Ugly: Pitfalls and Best Practices in Automated Sound Static Analysis of Ethereum Smart Contracts. In *International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020*, pages 212-231. Springer, 2020.

[SGSM20] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *27th ACM Conference on Computer and Communications Security, CCS 2020*. ACM 2020.

Contents

CHAPTER 1

Introduction

Cryptocurrencies emerged in the last ten years as a new, groundbreaking technology that enables distributed consensus in the presence of financial incentives. Cryptocurrencies[1] promise openly accessible systems for fully distributed money transfers and computation whose correctness can be publicly verified. Starting with the release of Bitcoin [Nak08] in 2008, cryptocurrencies currently have a market capitalization of over 300 billion dollars[2].

This economic interest in cryptocurrencies has been driving the development of many new applications. However, this fast-paced development has, in most cases, been lacking solid theoretical foundations. As a result, many deployed applications show severe security issues, which in the context of cryptocurrencies can immediately lead to monetary losses or a loss in the privacy of the users. Such security issues do not only constitute a threat to the affected parties but, beyond that, generally undermine the users' trust in the individual application and also in cryptocurrencies as a whole. An example of this is the famous DAO hack [the16] in the cryptocurrency Ethereum, which caused a 20% decrease in the currency's value.

To counter this trend, it is crucial to develop solid theoretical foundations for cryptocurrencies and to use these foundations to analyze the security of existing systems, enhance the security of these systems and create new systems that are secure by design.

In this thesis, we will focus on two main challenges in the domain of cryptocurrencies, 1) the security of Ethereum smart contracts and 2) the security of off-chain protocols:

1. The blockchain consensus mechanism lays the ground for arbitrary, distributed computation. This, in particular, allows users to implement advanced distributed applications in the form of reactive programs, so-called *smart contracts* when supported by the underlying system.

[1] We in this thesis focus on such cryptocurrencies built on permissionless blockchains and hence will refer with the term *cryptocurrency* to such systems.

[2] As of October 25th, 2020. Numbers are taken from `https://coinmarketcap.com`.

Ethereum [Woo14a] was the first cryptocurrency to come with a (quasi[3]) Turing complete language to support the development of such smart contracts. As a consequence, a versatile environment of applications emerged ranging from trading platforms [NGW17, MM17], over elections [MSH17] to auctions [HSLC17, GY18], just to mention a few examples. Using Ethereum as an openly accessible distributed computation platform, however, also comes with high-security risks. The integrity of Ethereum smart contracts is enforced as part of the consensus that makes applications immutable once deployed on the blockchain and their execution effects (which involve money transfers) irreversible. This does not only amplify the effects of programming bugs but also incentivizes attackers to develop malicious smart contracts to deceive honest users. It is the subject of ongoing research efforts in the community to answer the question of how to create an environment for jointly executed smart contracts that at the same time satisfies the practical security needs of the users.

2. The consensus mechanism of Ethereum and Bitcoin [Nak08] suffers from substantial performance problems. This manifests in a low transaction throughput which stays way behind banking systems such as Visa[4]. In addition to that, Bitcoin and Ethereum require the consensus parties to store the full transaction history of the system what makes the extensive use of these systems in the long run increasingly costly for their users in terms of storage consumption. One way to approach these issues is to reduce the usage of the blockchain-based consensus mechanism in favor of light-weight cryptographic peer-to-peer protocols that only rely on the blockchain as a safe fallback mechanism. In this way, the transaction load can be substantially reduced since the blockchain is (after an initial set-up) only used in case of a dispute. However, such *layer-two* or *off-chain* approaches introduce another layer of complexity and, with that, new security risks. For maintaining the security as given by the on-chain enforcement, the replacing off-chain protocols need to be carefully designed and proven correct.

1.1 Security Issues in Distributed Blockchain Applications

In the following, we will discuss concrete security issues that arise in distributed blockchain applications. We overview the cases of Ethereum smart contracts, where security issues mainly arise from the semantic subtleties of Ethereum's built-in program execution engine and of off-chain protocols (in Bitcoin), where security vulnerabilities are introduced at the level of cryptographic protocols built on top of the blockchain.

[3] Supporting a Turing complete instruction set, Ethereum enforces termination by bounding the number of computation steps based on a prespecified resource limit.

[4] Visa supports peaks of up to 47,000 transactions per second[vis] as compared to 7 transactions per second in Bitcoin and 13 transactions per second in Ethereum (Information is taken from `https://ycharts.com`).

1.1.1 Ethereum Smart Contracts

The joint execution of Ethereum smart contracts is subject to the so-called *Code is law* principle[5]: Once a smart contract is deployed (which corresponds to a transaction that creates a contract being appended to the blockchain), its code will be executed according to the *Ethereum Virtual Machine (EVM)*, a priorly agreed execution environment that is part of the consensus. Since the execution rules of the EVM are unambiguous, the execution behavior of a smart contract is well-defined, and hence it is expected to show the behavior as intended by the creator.

However, in practice, the behavior of smart contracts often deviates from the programmers' intentions what can be traced back to several peculiarities in the smart contract execution behavior and the way that smart contracts are developed. Smart contracts do not behave like sequential programs but need to be considered agents that interact with a largely unknown and potentially hostile environment with which they share state. To support environment interaction, *EVM bytecode*, the native low-level programming language of Ethereum smart constructs, supports special instructions, e.g., accessing blockchain data or calling other contracts. However, due to its low-level Assembly-like nature, smart contracts are usually not developed in EVM bytecode, but in other high-level languages (most prominently *Solidity* [sol19]) and then compiled to the bytecode format. These high-level languages aim at mimicking well-known programming languages from the sequential domain (such as *JavaScript*, *Java*™, or *Python*) in order to make smart contract programming accessible to a wide range of users. This obfuscates the complex, interactive nature of smart contracts and hence makes it even harder for developers to create correct contracts. As a result, in practice, one can observe significant gaps between the users' intuitions about smart contract semantics and the real behavior that a contract shows during execution on the blockchain.

In conjunction with the trend towards the development of more evolved smart contracts, this results in a toxic situation for the security of the users of the Ethereum system:

- Ethereum smart contracts are particularly security-critical since they control real money flows.
- Ethereum smart contracts are particularly prone to errors since they are usually complex and developed in languages that do not assist the design of secure smart contracts.
- Ethereum smart contract bugs are persistent since they cannot be patched once a program is deployed on the blockchain.

These are not only theoretical threats but practical problems, as demonstrated by infamous hacks, such as the DAO hack [the16] or the Parity hacks [par17a, par17b] which caused losses of several millions of dollars. To illustrate how the previously described factors facilitate real-work attacks, we will in the following describe the essence of the DAO hack where an attacker in 2016 stole

[5]The term 'code-is-law' was initially introduced by the law professor Lawrence Lessing [Les99] for characterizing the rules of cyberspace and their social implications. Nowadays, it is heavily adopted by the blockchain community to characterize the self-enforcing and immutable character of smart contracts.

```
1 contract DAO{
2   mapping (address => uint) bal;
3
4   function invest () public payable {
5     bal[msg.sender]+= msg.value;}
6
7   function withdraw () public {
8     address a = msg.sender;
9     if (bal[a] > 0){
10      a.call.value(bal[a]);
11      bal[a] = 0;}}
12 }
```

```
1 contract Mallory{
2  address DAO_ADDRESS = 0x...;
3  DAO dao = DAO(DAO_ADDRESS);
4
5  function investSmallAmount() public {
6  dao.invest().value(1);}
7
8  function() paybable{
9  dao.withdraw();}
10 }
```

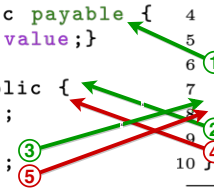

Figure 1.1: Simplified DAO contract.

more than 60 Million dollars, and that resulted in a hard fork - a change in consensus - on the Ethereum blockchain.

The DAO Hack. The DAO (short for *decentralized autonomous organization*) contract was a smart contract implementing a form of venture capital fund that allowed users to invest money and later retrieve back their investments. Due to a programming bug in the withdrawal functionality of the contract, an attacker could retrieve all investments made by other users. We illustrate the main workings of this attack with a simplified example in Figure 1.1. The depicted contracts are written in the high-level language *Solidity*. *Solidity* is developed by the Ethereum foundation and features a syntax that is inspired by *JavaScript*. Following the paradigm of object orientation, *Solidity* uses the concept of a contract analogously to the notion of a class in object-oriented programming. Contracts in Ethereum are entities that hold code and persistent storage containing the content of global contract fields such as the `bal` field of the `DAO` contract. This field here is a mapping from entity addresses to respective investments made by these entities. In Ethereum, all entities (users as well as smart contracts) are identified by 160-bit addresses that, in the case of users, correspond to cryptographic public keys. Apart from that, no explicit distinction between users and contracts is made, but users are considered entities without contract code. Consequently, users can (using their private key) initiate and authorize transactions to other entities. Such transactions also trigger the execution of the entity's code – which will have no effect in users' case but will initiate code execution in the case of smart contracts. To this end, transactions can also specify an input that is accessible by the triggered smart contract code. In *Solidity* this input is used to encode the distinct function to be executed and the arguments to the function.

It is important to note that one consequence of this architecture is that in Ethereum there is no clear distinction between the invocation of a contract function and the transfer of money. In particular, every contract invocation can potentially transfer money, and every money transfer can potentially trigger the execution of code. To this end *Solidity* supports the concept of a *fallback* function, a function without names or arguments (as present in the contract `Mallory`). This function contains the code that will get executed if the invocation of a contract (potentially due to a money transfer) does not specify a valid contract function to be executed. To further control the way that contracts can be invoked, *Solidity* provides modifiers that restrict the accessibility of functions: The modifier `public` indicates that the function can be invoked by any other user

or contract (as opposed to `private` functions that can only be invoked by the contract itself). The modifier `payable` denotes that money can be transferred along with the invocation of the function. Functions that are not explicitly marked to be `payable` will immediately throw an exception when being invoked with a positive amount of money.

The simple `DAO` contract depicted in Figure 1.1 supports two main functionalitiies: Users can invest money (`invest()`) and withdraw previously invested money (`withdraw()`). To this end, whenever the `invest()` function is invoked, the money transferred along with the invocation (accessible by the special variable `msg.value`) is recorded in the mapping `bal` for the investing entity (identified by the address of the sender as accessible by `msg.sender`). When a user calls the `withdraw()` function, the money as recorded in the `bal` mapping will be transferred back to the sender, and the `bal` mapping will be updated accordingly. To initiate the money transfer *Solidity*'s `call` construct is used. This construct reflects the basic contract interaction mechanism as supported by the low-level EVM bytecode. This mechanism does not distinguish between contract invocation and money transfer but simply initiates a transaction to the entity identified by the address `msg.sender` with the amount `bal[a]`.

This semantic subtlety can be exploited by an attacker that deploys a malicious contract `Mallory` to first make a small investment to the `DAO` contract (①) that they later withdraw (②). When the `withdraw()` function of the `DAO` contract calls back to the sender (`Mallory`, ③), not only the corresponding amount of Ether is transferred, but also the fallback function of `Mallory` is executed. `Mallory` implements this function to call the `DAO`'s `withdraw()` function (④), thereby *reentering* the `DAO` contract. Since at this point the balance of `Mallory` in the `bal` mapping has not been updated yet, another value transfer to `Mallory` will be initiated (⑤). By proceeding in this way, `Mallory` can drain all funds of the `DAO` contract.

The depicted attack is an example of how standard intuitions from (sequential) programming do not apply to smart contracts: In Ethereum, one needs to consider that a contract invocation hands over the control to a (partly) unknown environment that can potentially schedule arbitrary contract invocations – even before the original contract execution was completed. This is since smart contracts are system entities that can be referenced by their addresses and hence are not necessarily known at the point of contract creation. This non-standard feature is particularly hard to handle for programmers since Ethereum smart contracts lack a clear distinction between money transfers and code invocation. While this can be considered a questionable decision in the design of the Ethereum virtual machine, the resulting issues are aggravated by the fact that the *Solidity* language does not properly reflect these particularities of the semantics but instead tries to employ standard intuitions from sequential programming.

The example of the DAO illustrates well how the combination of programmers with lacking domain knowledge, a non-standard execution model, and non-suitable programming languages can lead to severe security flaws that have, in the context of cryptocurrencies, far-reaching consequences. Following the consensus mechanism and the resultant *Code is law* principle, the extensive money theft conducted during the DAO attack should have been irreversible. However, this incident initiated a heated discussion in the Ethereum community and finally resulted in a hard fork of the Ethereum blockchain [har16]: A majority of the users agreed to change the consensus rules such that the effects of the DAO attack were ignored. This technically led to

a split of Ethereum into two currencies, one following the old consensus rules (including the effects of the DAO), called today *Ethereum Classic*, and another currency (that is still called *Ethereum*) that follows the new consensus rules. Such hard forks sustainably erode the trust in systems like Ethereum because they undermine the main security promise of blockchain-based cryptocurrencies to enforce upfront agreed consensus rules.

To prevent such incidents, it is hence of paramount importance to provide the users of Ethereum with infrastructure and tools that allow them to ensure that the smart contracts that they create and that they interact with show the behavior that they expect. To this end, it is crucial to generically characterize the dangerous behavior of smart contracts that users might want to prevent and to enable the users to automatically and reliably check for the absence of such behavior.

1.1.2 Off-chain Protocols

In contrast to Ethereum, the focus of Bitcoin is on implementing a distributed monetary system. To this end, Bitcoin supports a very limited scripting language that is used to express conditions on the validity of a financial transaction. This focus on financial transactions urges the comparison with centralized payment systems, such as Visa. However, this comparison highlights the scalability issues of Bitcoin (and other cryptocurrencies that rely on similar consensus mechanisms): The transaction throughput of Bitcoin is limited to approximately 7 transactions per second, as opposed to Visa that can handle peaks up to 47,000 transactions per second[6].

This is an inherent limitation to Bitcoin's proof of work (PoW) consensus mechanism that relies on users to solve computationally hard puzzles to include new transactions and thereby advance the system. This procedure is not only time-consuming as such but adds an additional time overhead since the system requires some time before converging towards a stable state in which a transaction can be considered complete.

Further, cryptocurrencies such as Bitcoin require that all users of the system save the whole transaction history in order to verify the validity of new transactions. With a growing number of transactions, this becomes a non-negligible and constantly deteriorating burden for the users that hinders the accessibility of such cryptocurrencies.

These problems limit the potential of Bitcoin (and similar cryptocurrencies) to scale to a large number of users and transactions in the long run. To still enable the large-scale deployment of such cryptocurrencies, it is crucial to find principled solutions to these problems that ideally do not require changes to the consensus of the deployed systems to avoid hard forks. In contrast to several proposals that rely on a change in the consensus [LNZ+16, BGM16] (so called *layer-one* solutions), *payment channel networks (PCNs)* promise to alleviate scalability issues while staying compatible with existing cryptocurrencies. [7]

[6] See Footnote 4 This limititation, however, only affects the transaction throughput, not the the transaction volume.

[7] Even though PCNs allow for an enormous increase in the transaction throughput, they come with other limitations. In particular, they require users to (temporarily) lock funds on-chain and by this hinder large transaction volumes. For a short comparison with a recently proposed layer-two solution, so-called *rollups*, that does not come with this limitation, we refer the reader to Section 6.2.

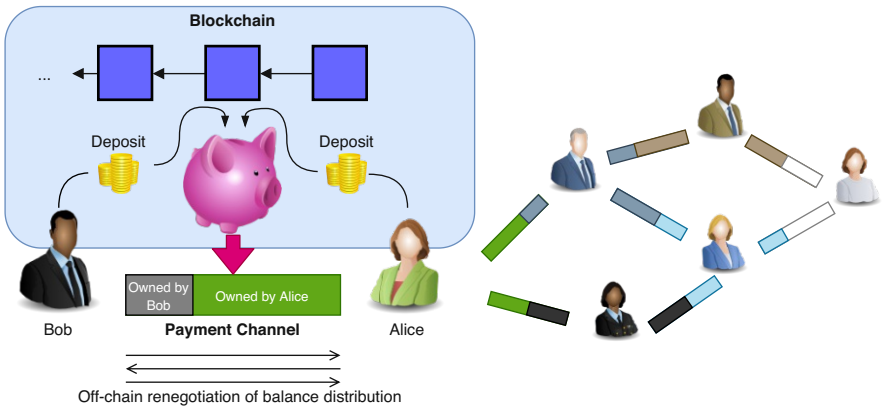

Figure 1.2: Illustration of the workings of payment channel networks

Payment Channel Networks. The general idea of PCNs is that it should be possible to perform simple payments between honest users in a peer-to-peer manner without recording each of these transactions individually on the blockchain. Instead, the blockchain should only be leveraged to resolve conflicts in the case of a dispute. This would significantly lower the transaction load on the blockchain and hence mitigate the scalability problems.

The key concept of PCNs is two-party *payment channels* which allow for such simple money transfers between the involved parties. We illustrate the main workings of payment channels in Fig. 1.2: The parties (here Alice and Bob) deposit a certain amount of money on the blockchain and then negotiate in a peer-to-peer protocol the ownership distribution of the deposited money. Payments between the parties then correspond to the renegotiation of the ownership distribution. In the process of renegotiation, the involved parties exchange information that makes the outcome of the negotiation enforceable on the blockchain so that in case that a party stops collaborating, the other party can publish a transaction that pays out the deposits according to the last agreed-upon distribution.

Since payment channels require deposits and temporarily lock the money of users, it is not feasible for users to maintain payment channels with all other users with whom they (potentially) want to exchange money. In order to solve this problem, payment channels are extended to PCNs in which payments can be routed through intermediate users that connect the sender and the receiver via (two-party) payment channels.

However, performing such *multi-hop payments* in a PCN, requires to use an involved multi-party protocol: It needs to be ensured that intermediate users 1) cannot steal the transferred money 2) cannot lose money, and 3) have an incentive to participate in the payment. This is particularly challenging since it always needs to be taken into account that users may stop collaborating (or simply go offline).

To illustrate the security issues that can arise in PCNs, we in the following describe the *wormhole attack* on Bitcoin's Lightning Network [PD] that we recently discovered.

Wormhole Attack. For performing atomic multi-hop payments in the Lightning Network, the

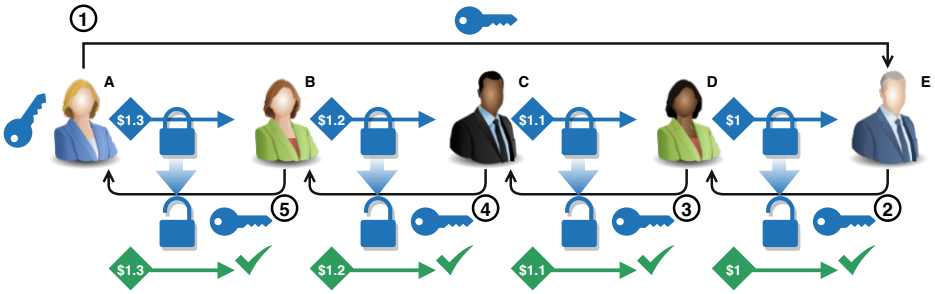

Figure 1.3: Illustration of an honest payment in the Lightning Network

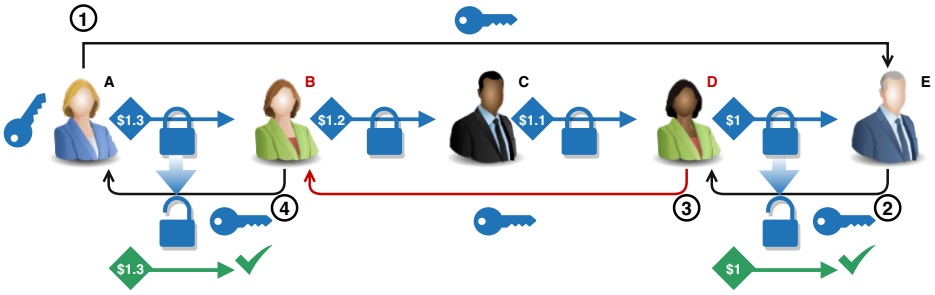

Figure 1.4: Wormhole attack on the Lightning Network

intermediate users along a payment path first *lock* the amount to be transferred in their respective pairwise payment channels. Locking in this context means that the channel parties renegotiate the channel balance in a way that only is enforceable once a specific secret (the key) is learned. In the Lightning Network, the key corresponds to the preimage of a hash value that is chosen by the sender of the payment. Fig. 1.3 illustrates such a multi-hop payment from A to E using the parties B, C, and D as intermediaries. Once all parties along the path locked their corresponding coins, the sender (here A) releases the key to the receiver (here E) of the payment (step ①). With the knowledge of the key, the receiver E can unlock the conditional payment from D (step ②). In this process, D learns the key that enables them to unlock them payment from C, and so on, until the payment from A to B is unlocked (step ⑤), which concludes the payment.

In order to account for network failures and malicious parties, the payments in the channels are only temporarily locked and will be released after a predefined timeout. Consequently, if, e.g., the intermediate user C does not collaborate in the unlocking phase, the payment to D can still be enforced on the blockchain, resulting in a payment from D to E, while the payment from A to C will be rolled back due to the timeout.

To incentivize the participation of intermediate users, payment senders add small payment fees to the payments that can be kept by the intermediaries when performing the payment. As shown in Fig. 1.3, A conducts a payment of 1.3 coins, out of which 1 coin will be transferred to E while the remaining 0.3 coins stay with the intermediate users when forwarding the coins.

The wormhole attack enables malicious intermediate users to steal these payment fees from other

users on the path. We illustrate this in Fig. 1.4 where the parties B and D collude in order to steal the payment fees of C. To this end, D in the unlocking phase, instead of unlocking the payment from C, forwards the key that they learned from E in step ② to party B (step ③) who will conclude the payment with A (step ④). For party C, the payment will timeout, resulting in C rolling back the forwarded payment and not obtaining any fee, as it would happen in the case of a failed payment. However, the payment was successful, and B and D in sum gained 0.3 coins corresponding to their own and the payment fees meant for C.

As a consequence of this attack, honest intermediate users lose their incentive to participate in multi-hop payments. Since they can be potentially deprived of their payment fee, there is no reason for honest users to participate as intermediaries in a payment that requires them to lock some of their coins for a certain amount of time and hence comes with collateral costs. The wormhole attack is particularly problematic as a victim of the attack cannot distinguish between the attack scenario and the scenario where the payment failed for legitimate reasons (e.g., because the receiver went offline). Consequently, there is no way of holding malicious parties accountable for the attack.

Attacks as the wormhole attack are particularly problematic since they undermine the effectiveness of PCNs that rely on the broad participation of the users of the cryptocurrency to create a well-connected network graph and on the willingness of the users to forward payments in this graph.

1.2 Methodology

The presented challenges illustrate the variety of distributed applications in the context of cryptocurrencies. While smart contracts in Ethereum allow for the encoding of arbitrary functionality that is immediately enforced as part of the consensus mechanism, the example of PCNs motivates the significance of building applications in the form of cryptographic protocols which only rely indirectly on the blockchain's enforcement mechanism. Both forms of applications have in common that reliable security guarantees are of paramount importance to enable the safe participation of the application's users. Simultaneously, the distributed blockchain environment brings particular challenges to the safe implementation of these applications and increases the potential for security-critical mistakes. Still, principled theoretical foundations for these applications are in large parts missing, and in particular, this applies to well-defined security notions. For solving this situation, it is required to

- Rigorously formalize the workings of distributed applications in cryptocurrencies.
- Formally characterize relevant security properties for distributed applications.
- Design distributed applications that satisfy the formal security properties.
- Give formal proofs for the security of distributed applications.
- Design tools to analyze the security of distributed applications.

1.3 Contributions

This thesis aims at providing theoretical foundations for the security of distributed blockchain applications in order to tackle the previously described challenges. To this end, we formalize the workings of distributed applications and define formal security and privacy properties for them. We use the developed security notions to pinpoint issues in existing applications and show how to systematically enhance their security by fixing existing problems, designing tools for formal security proofs, and finally conducting such formal proofs.

1.3.1 Semantic Foundations for Ethereum Smart Contracts

In Chapter 2 we lay the semantic foundations for Ethereum smart contracts by introducing a complete small-step semantics for EVM bytecode that is accompanied by a formalization in the proof assistant F*.

The F* implementation allows us (via prior compilation to OCaml) to validate the presented semantics against the official Ethereum test suite and hence to reliably link our formal semantics to the existing client implementations of the EVM, which constitute the consensus rules of Ethereum.

Further, with the F* implementation, we provide a framework for machine-checked proofs that can be used to prove properties of individual smart contracts or to reason about the correctness of analysis techniques for smart contract verification.

Based on the introduced semantics, we give the first formal definitions of crucial security properties for smart contracts. In particular, we introduce the notion of *call integrity* that characterizes the robustness of a contract's call behavior against the influence of unknown contracts. This generic property also rules out reentrancy attacks such as the one on the DAO contract. We further devise a dedicated proof technique for call integrity, and in this course, formulate the notion of *single-entrancy* that more specifically characterizes the restrictions on reentering execution behavior needed to avoid reentrancy attacks. Further, we introduce the notions of *atomicity*, and *independence from miner controlled parameters*, properties that are all motivated by real-world attacks on Ethereum smart contracts.

1.3.2 Trends and Challenges in the Security Analysis of Ethereum Smart Contracts

In Chapter 3 we overview the existing approaches taken towards formal verification of Ethereum smart contracts.

For this purpose, we survey recent theories and tools for formal verification of Ethereum smart contracts, including a systematization of existing work with an overview of the open problems and future challenges in the smart contract realm.

Specifically, we discuss the domain-specific challenges that arise in the design of *automated sound* static analysis that so far hindered the development of such analyzers. These challenges can be mainly attributed to the particular language design of EVM bytecode (such as a statically

unknown control graph and a non-standard memory layout) and the peculiarities of the distributed blockchain execution environment (which features many statically unknown components, in particular, other interacting contracts).

To illustrate the difficulties and pitfalls in this domain, we overview existing automated analyzers that aim at giving soundness guarantees. We highlight the individual difficulties by giving examples of unsoundness introduced by these tools in the different stages of the analysis process.

1.3.3 *eThor*: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts

In Chapter 4 we present *eThor*, the first provably sound static analyzer for EVM bytecode.

The core of *eThor* is a static reachability analysis that soundly abstracts the small-step semantics of EVM bytecode as Horn clauses. We show this reachability analysis to be sufficient to verify relevant generic security properties, in particular the single-entrancy property presented in Chapter 2 that ensures robustness against reentrancy attacks. Further, we illustrate how to use *eThor* for verifying contract-specific functional properties that can be expressed in the form of pre- and postconditions. We highlight how *eThor* overcomes the domain-specific challenges discussed in Chapter 3, in particular how *eThor* soundly abstracts the interaction with statically unknown contracts while still maintaining reasonable precision.

We provide a formal soundness proof that shows the static reachability analysis of *eThor* to approximate the formal semantics of EVM bytecode presented in Chapter 2 soundly. Further, we show how single-entrancy and other relevant properties can be soundly abstracted in terms of the reachability analysis.

For developing a robust and practical analyzer that closely follows the reachability analysis as covered by the soundness proof, we introduce *HoRSt*, a general framework to specify static analyses based on Horn clauses and to automatically generate performant analysis tools from these high-level specifications. For this purpose, *HoRSt*, on input of the high-level analysis specification, produces an optimized `smt-lib` [smt20] encoding suitable for *z3* [HB12].

We show how to use *HoRSt* for implementing the static analysis tool *eThor*. To systematically evaluate *eThor*, we first experimentally confirm its soundness and assess its precision on the official EVM tests that we can automatically encode as functional contract properties. Next, we showcase *eThor*'s performance by evaluating it against the state-of-the-art analyzer ZEUS [KGDS18a] on the data set of real-world contracts presented in [KGDS18a]. Even though ZEUS claims to be sound, in our evaluation, ZEUS shows a striking specificity (i.e., completeness) of 99.8%, but a recall of only 11.4% – which empirically refutes ZEUS' soundness claim. In contrast, *eThor* still achieves a specificity of 80.4% while being sound (hence having a recall of 100%). Consequently, *eThor* with an F-measure of 89.1%, shows a significantly better overall performance than ZEUS with an F-measure of 20.4%.

1.3.4 Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability

In Chapter 5 we analyze the security of existing PCNs and devise a novel cryptographic primitive that captures the core security and privacy requirements for safe payments in PCNs.

In particular, we report a new attack on PCNs (the *wormhole attack*), enabling attackers to steal payment fees from honest users in a multi-hop payment. Since payment fees serve as an incentive for users to participate as intermediate users in multi-hop payments, this attack undermines the motivation of honest users to take part in payments in the first place. We could show this attack not only to apply to Bitcoin's Lightning Network but to be an inherent limitation of PCNs (following the definition in [MMSK+17]) where the sender does not know the intermediate users along the path to the receiver.

To systematically overcome this security limitation, we introduce a new cryptographic primitive called *anonymous multi-hop lock* (AMHL). We give a concise characterization of the primitive's security requirements in the UC framework [Can01] and show how this allows for using AMHL as a building block in the construction of secure PCNs based on blockchains.

The AMHL primitive assumes the sender's knowledge of the payment path and an initial communication phase between the sender and the intermediate users. We prove this additional round of communication necessary for avoiding the wormhole attack by establishing a lower bound on the communication complexity of secure PCNs that follow the definition from [MMSK+17].

Finally, we show how to realize the AMHL primitive in practice. To this end, we demonstrate that a AMHL can be realized in any cryptocurrency whose scripting language supports (linearly) homomorphic one-way functions. Further, we show that AMHLs can also be realized in scriptless settings – without making use of a specific scripting language, but by encoding into the transaction signatures. This approach is of special interest because it reduces the transaction size and, consequently, the blockchain load and at the same time ensures the fungibility of the currency, making payment channel transactions indistinguishable from other transactions. We present a concrete construction for realizing AMHLs with the ECDSA signature scheme. Since this scheme is used for transaction signatures in most prominent cryptocurrencies, this construction makes AMHLs compatible with the vast majority of cryptocurrencies (including Bitcoin and Ethereum).

As a consequence, AMHLs have already been implemented and tested in Bitcoin's Lightning Network [Froa, Frob].

CHAPTER 2

Semantic Foundations for Ethereum Smart contracts

Abstract

Smart contracts are programs running on cryptocurrency (e.g., Ethereum) blockchains, whose popularity stems from the possibility to perform financial transactions, such as payments and auctions, in a distributed environment without the need for any trusted third party. Given their financial nature, bugs or vulnerabilities in these programs may lead to catastrophic consequences, as witnessed by recent attacks. Unfortunately, programming smart contracts is a delicate task that requires strong expertise: Ethereum smart contracts are written in *Solidity*, a dedicated language resembling *JavaScript*, and shipped over the blockchain in the EVM bytecode format. In order to rigorously verify the security of smart contracts, it is of paramount importance to formalize their semantics as well as the security properties of interest, in particular at the level of the bytecode being executed.

In this chapter, we present the first complete small-step semantics of EVM bytecode, which we formalize in the F* proof assistant, obtaining executable code that we successfully validate against the official Ethereum test suite. Furthermore, we formally define for the first time central security properties for smart contracts, such as call integrity, atomicity, and independence from miner-controlled parameters. This formalization relies on a combination of hyper- and safety properties. Along with this work, we identified various mistakes and imprecisions in existing semantics and verification tools for Ethereum smart contracts, thereby demonstrating once more the importance of rigorous semantic foundations for the design of security verification techniques.

This chapter presents the first result of the collaboration with Ilya Grishchenko and Matteo Maffei and was published at the 7th International Conference on Principles of Security and Trust (POST'18) under the title 'A Semantic Framework for the Security Analysis of Ethereum Smart Contracts' [GMS18b]. It received the EAPLS best paper award of the umbrella conference ETAPS.

I am responsible for the formalization of the small-step semantics and the security properties, as well as the corresponding proofs. The F⋆ implementation was mostly done by Ilya Grishchenko. For better illustration of the semantics, figures, and explanations from our invited contribution 'Foundations and Tools for the Static Analysis of Ethereum Smart Contracts' [GMS18a] to the 30th International Conference on Computer Aided Verification (CAV 2018) were added to the background sections. The accompanying appendix contains extended versions of the formalism and proofs, as well as an updated version of the semantics that accounts for recent changes in the Ethereum smart contract semantics that were introduced after the publication of the paper.

2.1 Introduction

One of the determining factors for the growing interest in blockchain technologies is the groundbreaking promise of secure distributed computations, even in the absence of trusted third parties. Building on a distributed ledger that keeps track of previous transactions and the state of each account, whose functionality and security is ensured by a delicate combination of incentives and cryptography, software developers can implement sophisticated distributed, transaction-based computations by leveraging the scripting language offered by the underlying cryptocurrency. While many of these cryptocurrencies have an intentionally limited scripting language (e.g., Bitcoin [Nak08]), Ethereum was designed from the ground up with a quasi Turing complete language[1]. Ethereum programs, called *smart contracts*, have thus found a variety of appealing use cases, such as financial contracts [BKT17], auctions [HSLC17], elections [MFSH17], data management systems [Adh17], trading platforms [NGW17, MM17], permission management [AEVL16] and verifiable cloud computing [DWA+17], just to mention a few. Given their financial nature, bugs and vulnerabilities in smart contracts may lead to catastrophic consequences. For instance, the infamous DAO vulnerability [the16] recently led to a 60M$ financial loss, and similar vulnerabilities occur on a regular basis [par17a, par17b]. Furthermore, many smart contracts in the wild are intentionally fraudulent, as highlighted in a recent survey [ABC17].

A rigorous security analysis of smart contracts is thus crucial for the trust of the society in blockchain technologies and their widespread deployment. Unfortunately, this task is quite challenging for various reasons. First, Ethereum smart contracts are developed in an ad-hoc language, called *Solidity*, which resembles *JavaScript* but features specific transaction-oriented mechanisms and a number of non-standard semantic behaviors, as further described in this chapter. Second, smart contracts are uploaded on the blockchain in the form of Ethereum Virtual Machine (EVM) bytecode, a stack-based low-level code featuring dynamic code creation and invocation and, in general, very little static information, which makes it extremely difficult to analyze.

Related Work Recognizing the importance of solid semantic foundations for smart contracts, the Ethereum foundation published a yellow paper [Woo14b] to describe the intended behavior of smart contracts. This semantics, however, exhibits several under-specifications and does not follow any standard approach for the specification of program semantics, thereby hindering

[1] While the language itself is Turing complete, computations are associated with a bounded computational budget (called gas), which gets consumed by each instruction thereby enforcing termination.

program verification. In order to provide a more precise characterization, Hirai formalizes the EVM semantics in the proof assistant Isabelle/HOL and uses it for manually proving safety properties for concrete programs [Hir17a]. This semantics, however, constitutes just a sound over-approximation of the original semantics [Woo14b]. More specifically, once a contract performs a call that is not a self-call, it is assumed that arbitrary code gets executed. Consequently, arbitrary changes to the account's state and to the global state can be performed. Consequently, this semantics can not serve as a general-purpose basis for static analysis techniques that might not rely on the same over-approximation.

Hildebrandt et al. [HSR+18] define the EVM semantics in the $\mathbb{K}$ framework [SPY+16] – a language-independent verification framework based on reachability logics. The authors leverage the power of the $\mathbb{K}$ framework in order to automatically derive analysis tools for the specified semantics, presenting as an example a gas analysis tool, a semantic debugger, and a program verifier based on reachability logics. The underlying semantics relies on local rewriting rules on the system configuration. Since parts of the execution are treated in separation, such as the exception behavior and the gas calculations, one small-step consists of several rewriting steps, which makes this semantics harder to use as a basis for new static analysis techniques. This is relevant whenever the static analysis tools derivable by the $\mathbb{K}$ framework are not sufficient for the desired purposes: for instance, their analysis requires the user to manually specify loop invariants, which is hardly doable for EVM bytecode and clearly does not scale to large programs. In recent work [PZS+18], the proposed $\mathbb{K}$ semantics is leveraged to generate a deductive verifier that incorporates domain-specific abstractions for EVM bytecode in order to give improved performance. Still, all these works concentrate on the semantics of EVM bytecode but do not study security properties for smart contracts.

Sergey et al. [SH17] compare smart contracts on the blockchain with concurrent objects using shared memory and use this analogy to explain typical problems that arise when programming smart contracts in terms of concepts known from concurrency theory. They encourage the application of state-of-the -rt verification techniques for concurrent programs to smart contracts but do not describe any specific analysis method applied to smart contracts themselves. Mavridou et al. [ML18] define a high-level semantics for smart contracts that is based on finite state machines and aims at simplifying the development of smart contracts. They provide a translation of their state machine specification language to *Solidity*, a higher-order language for writing Ethereum smart contracts, and present design patterns that should help users to improve the security of their contracts. The translation to *Solidity* is not backed up by a correctness proof, and the design patterns are not claimed to provide any security guarantees.

Bhargavan et al. [BDLF+16b] introduce a framework to analyze Ethereum contracts by a translation into F*, a functional programming language aimed at program verification and equipped with an interactive proof assistant. The translation supports only a fragment of the EVM bytecode and does not come with a justifying semantic argument.

Luu et al. have recently presented Oyente [LCO+16a], a state-of-the-art static analysis tool for EVM bytecode that relies on symbolic execution. Oyente comes with a semantics of a simplified fragment of the EVM bytecode and, in particular, misses several important commands related to contract calls and contract creation. Furthermore, it is affected by a major bug related to

calls as well as several other minor ones, which we discovered while formalizing our semantics, which is inspired by theirs. Oyente supports a variety of security properties, such as transaction order dependency, timestamp dependency, and reentrancy, but the security definitions are rather syntactic and described informally. As we show in this chapter, the lack of solid semantic foundations causes several sources of unsoundness in Oyente.

Our Contributions This chapter lays the semantic foundations for Ethereum smart contracts. Specifically, we introduce

- The first complete small-step semantics for EVM bytecode;
- A formalization in F* of a large fragment of our semantics, which can serve as a foundation for verification techniques based on encoding into this language [BDLF+16b] as well as machine-checked proofs for other analysis techniques (e.g., [LCO+16a]). By compiling F* to OCaml, we could successfully validate our semantics against the official Ethereum test suite;
- The first formal definitions of crucial security properties for smart contracts, such as call integrity, for which we devise a dedicated proof technique, atomicity, and independence from miner-controlled parameters. Interestingly enough, the formalization of these properties requires hyperproperties, while existing static analysis techniques for smart contracts rely on reachability properties and syntactic conditions;
- A collection of examples showing how the syntactic conditions employed in current analysis techniques are imprecise and, in several cases, unsound, thereby further motivating the need for solid semantic foundations and rigorous security definitions for smart contracts.

The complete semantics, as well as the formalization in F*, are publicly available [GMS18c].

Outline The remainder of this chapter is organized as follows. Section 2.2 briefly overviews the Ethereum architecture, Section 2.3 introduces the Ethereum semantics and our formalization in F*, Section 2.4 formally defines various security properties for Ethereum smart contracts, and Section 2.5 concludes highlighting interesting research directions.

2.2 Background on Ethereum

Ethereum

Ethereum is a cryptographic currency system built on top of a blockchain. Similar to Bitcoin, network participants publish transactions to the network that are then grouped into blocks by distinct nodes (the so-called *miners*) and appended to the blockchain using a proof of work (PoW) consensus mechanism. The state of the system – that we will also refer to as *global state* – consists of the state of the different accounts populating it. An account can be either an external account (belonging to a user of the system) that carries information on its current balance

or a contract account that additionally obtains persistent storage and the contract's code. The account's balances are given in the subunit *wei* of the virtual currency *Ether*.[2]

Transactions can alter the system's state by either creating new contract accounts or by calling an existing account. Calls to external accounts can only transfer Ether to this account, but calls to contract accounts additionally execute the code associated with the contract. The contract execution might alter the storage of the account or might again perform transactions – we talk about *internal transactions* in this case.

The execution model underlying the execution of contract code is described by a virtual state machine, the *Ethereum Virtual Machine* (EVM). This is *quasi Turing complete* as the otherwise Turing complete execution is restricted by the upfront defined resource *gas* that effectively limits the number of execution steps. The originator of the transaction can specify the maximal gas that should be spent for the contract execution and also determines the gas price (the amount of wei to pay for a unit of gas). Upfront, the originator pays for the gas limit according to the gas price, and in case of successful contract execution that did not spend the whole amount of gas dedicated to it, the originator gets reimbursed with gas that is left. The remaining wei paid for the used gas is given as a fee to a beneficiary address specified by the miner.

EVM Bytecode

The code of contracts is written in *EVM bytecode* – an Assembler-like bytecode language. As the core of the EVM is a stack-based machine, the set of instructions in EVM bytecode consists mainly of standard instructions for stack operations, arithmetics, jumps, and local memory access. The classical set of instructions is enriched with an opcode for the SHA3 hash and several opcodes for accessing the environment that the contract was called in. In addition, there are opcodes for accessing and modifying the storage of the account currently running the code and distinct opcodes for performing internal call and create transactions. Another instruction particular to the blockchain setting is the SELFDESTRUCT code that deletes the currently executed contract - but only after the successful execution of the external transaction.

Gas and Exceptions The execution of each instruction consumes a positive amount of gas. There is a gas limit set by the sender of the transaction. Exceeding the gas limit results in an exception that reverts the effects of the current transaction on the global state. In the case of nested transactions, the occurrence of an exception only reverts its own effects, but not those of the calling transaction. Instead, the failure of an internal transaction is only indicated by writing zero to the caller's stack.

Solidity

In practice, most Ethereum smart contracts are not written in EVM bytecode directly, but in the high-level language *Solidity*, which is developed by the Ethereum Foundation [sol19]. For understanding the typical problems that arise when writing smart contracts, it is important to consider the design of this high-level language.

[2] One Ether is equivalent to 10^{18} wei.

Solidity is a so-called “contract-oriented" programming language that uses the concept of class from object-oriented languages for the representation of contracts. Similar to classes in object-oriented programming, contracts specify fields and methods for contract instances. Fields can be seen as persistent storage of a contract (instance), and contract methods can, by default, be invoked by any internal or external transaction. This can be restricted *modifiers*, most prominently `public` and `private` that cause an exception upon invocation if certain preconditions are not met. Functions with a `private` modifier do so whenever invoked from the outside. Functions that are not explicitly marked as `private` default to `public` and hence do not impose any restrictions. For interacting with another contract, one either needs to create a new instance of this contract (in which case a new contract account with the functionality described in the contract class is created), or one can directly make transactions to a known contract address holding a contract of the required shape. The syntax of *Solidity* resembles *JavaScript*, enriched with additional primitives accounting for the distributed setting of Ethereum. In particular, *Solidity* provides primitives for accessing the transaction and the block information, like `msg.sender` for accessing the address of the account invoking the method or `msg.value` for accessing the amount of *wei* transferred by the transaction that invoked the method.

Solidity shows some particularities when it comes to transferring money to another contract especially using the provided low-level functions `send` and `call`. A value transfer initiated using these functions is finally translated to an internal call transaction which implies that calling a contract might also execute code and, in particular, it can fail because the available gas is not sufficient for executing the code. In addition – as in the EVM – these kinds of calls do not enable exception propagation so that the caller manually needs to check for the return result. Another special feature of *Solidity* is that it allows for defining so-called *fallback functions* for contracts that get executed when a call via the `send` function was performed or (using the `call` function) an address is called that, however, does not properly specify the concrete function of the contract to be called.

2.3 Small-Step Semantics

We introduce a small-step semantics covering the full EVM bytecode, inspired by the one presented by Luu et al. [LCO+16a], which we substantially revise in order to handle the missing instructions, in particular contract calls and call creation. In addition, while formalizing our semantics, we found a major flaw related to calls and several minor ones (cf. Section 2.3.7), which we fixed and reported to the authors. We refer the reader to Appendix A.1 and Appendix A.2 for a formal account of the semantic rules and present below the most significant ones.

2.3.1 Preliminaries

In the following, we will use $\mathbb{B}$ to denote the set $\{0, 1\}$ of bits and accordingly $\mathbb{B}^x$ for sets of bitstrings of size x. We further let $\mathbb{N}_x$ denote the set of non-negative integers representable by x bits and allow for implicit conversion between those two representations. In addition, we will use the notation $[X]$ (resp. $\mathcal{L}(X)$) for arrays (resp. lists) of elements from the set X. We use standard notations for operations on arrays and lists.

2.3.2 Global state

As mentioned before, the global state is a (partial) mapping from account addresses (that are bitstrings of size 160) to accounts. In the case that an account does not exist, we assume it to map to $\bot$. Accounts, irrespectively of their type, are tuples of the form $(n, b, \mathit{stor}, \mathit{code})$, with $n \in \mathbb{N}_{256}$ being the account's nonce that is incremented with every other account that the account creates, $b \in \mathbb{N}_{256}$ being the account's balance in *wei*, $\mathit{stor} \in \mathbb{B}^{256} \to \mathbb{B}^{256}$ being the accounts persistent storage that is represented as a mapping from 256-bit words to 256-bit words and finally $\mathit{code} \in [\mathbb{B}^8]$ being the contract that is an array of bytes. In contrast to contract accounts, external accounts have the empty byte array as code. As only the execution of code in the context of the account can access and modify the account's storage, the fact that formally external accounts have persistent storage does not have any effect. In the following, we will denote the set of addresses with $\mathcal{A}$ and the set of global states with Σ, and we will assume that $\sigma \in \Sigma$.

2.3.3 Small-Step Relation

In order to define the small-step semantics, we give a small-step relation $\Gamma \vDash S \to S'$ that specifies how a call stack $S \in \mathbb{S}$ representing the state of the execution evolves within one step under the transaction environment $\Gamma \in \mathcal{T}_{\mathit{env}}$.

In Figure 2.1 we give a full grammar for call stacks and transaction environments:

Call stacks	$\mathbb{S}$	$\ni$	S	$:=$	$\mathit{EXC} :: S_{\mathit{plain}} \mid \mathit{HALT}(\sigma, g, d, \eta) :: S_{\mathit{plain}} \mid S_{\mathit{plain}}$
Plain call stacks	$\mathbb{S}_{\mathit{plain}}$	$\ni$	S_{plain}	$:=$	$(\mu, \iota, \sigma, \eta) :: S_{\mathit{plain}}$
Machine states	M	$\ni$	μ	$:=$	$(\mathit{gas}, \mathit{pc}, m, i, s)$
Execution environments	I	$\ni$	ι	$:=$	$(\mathit{actor}, \mathit{input}, \mathit{sender}, \mathit{value}, \mathit{code})$
Global states	Σ	$\ni$	σ		
Account states	$\mathbb{A}$	$\ni$	acc	$:=$	$(n, b, \mathit{code}, \mathit{stor}) \mid \bot$
Transaction effects	N	$\ni$	η	$:=$	$(b, L, S_\dagger)$
Transaction environments	$\mathcal{T}_{\mathit{env}}$	$\ni$	Γ	$:=$	(o, price, H)

Notations: $d \in [\mathbb{B}^8]$, $g \in \mathbb{N}_{256}$, $\eta \in N$, $o \in \mathcal{A}$, $\mathit{price} \in \mathbb{N}_{256}$, $H \in \mathcal{H}$
$\mathit{gas} \in \mathbb{N}_{256}$, $\mathit{pc} \in \mathbb{N}_{256}$, $m \in \mathbb{B}^{256}, \to \mathbb{B}^8$ $i \in \mathbb{N}_{256}$, $s \in \mathcal{L}(\mathbb{B}^{256})$
$\mathit{sender} \in \mathcal{A}$ $\mathit{input} \in [\mathbb{B}^8]$ $\mathit{sender} \in \mathcal{A}$ $\mathit{value} \in \mathbb{N}_{256}$ $\mathit{code} \in [\mathbb{B}^8]$
$b \in \mathbb{N}_{256}$ $L \in \mathcal{L}(\mathit{Ev}_{\mathit{log}})$ $S_\dagger \subseteq \mathcal{A}$ $\Sigma = \mathcal{A} \to \mathbb{A}$

Figure 2.1: Grammar for call stacks and transaction environments

Transaction Environments

The transaction environment represents the static information of the block that the transaction is executed in and the immutable parameters given to the transaction as the gas price or the gas limit. More specifically, the transaction environment $\Gamma \in \mathcal{T}_{\mathit{env}} = \mathcal{A} \times \mathbb{N}_{256} \times \mathcal{H}$ is a tuple of the form (o, price, H) with $o \in \mathcal{A}$ being the address of the account that made the transaction, $\mathit{price} \in \mathbb{N}_{256}$ denoting amount of wei that needs to paid for a unit of gas in this transaction and

$H \in \mathcal{H}$ being the header of the block that the transaction is part of. We do not specify the format of block headers here but assume a set $\mathcal{H}$ of block headers.

Call Stacks

A call stack S is a stack of execution states which represents the state of the execution within one internal transaction. We give a formal definition of the set of possible call stacks $\mathbb{S}$ as follows:

$$\begin{aligned}\mathbb{S} := \{&\mathit{EXC} :: S_{\mathit{plain}},\ \mathit{HALT}(\sigma, \mathit{gas}, d, \eta) :: S_{\mathit{plain}},\ S_{\mathit{plain}} \\ &\mid \sigma \in \Sigma,\ \mathit{gas} \in \mathbb{N},\ d \in [\mathbb{B}^8],\ \eta \in N\ , S_{\mathit{plain}} \in \mathcal{L}(M \times I \times \Sigma \times N)\}\end{aligned}$$

Syntactically, a call stack is a stack of regular execution states of the form $(\mu, \iota, \sigma, \eta)$ that can optionally be topped with a halting state $\mathit{HALT}(\sigma, \mathit{gas}, d, \eta)$ or an exception state EXC. We summarize these three types of states as execution states $\mathcal{S}$. Semantically, halting states indicate regular halting of an internal transaction, exception states indicate exceptional halting, and regular execution states describe the state of internal transactions in progress. Halting and exception states can only occur as top elements of the call stack as they represent terminated internal transactions. Exception states of the form EXC do not carry any information as in the case of an exception, all effects of the terminated internal transaction are reverted, and the caller state, therefore, stays unaffected, except for the gas. Halting states instead are of the form $\mathit{HALT}(\sigma, \mathit{gas}, d, \eta)$ specifying the global state σ the execution halted in, the gas $\mathit{gas} \in \mathbb{N}_{256}$ remaining from the execution, the return data $d \in [\mathbb{B}^8]$ and the additional transaction effects $\eta \in N$ of the internal transaction. The additional transaction effects carry information accumulated during execution but not influencing the small-step execution itself. Formally, the additional transaction effects are a triple of the form $(b, L, S_\dagger) \in N = \mathbb{N}_{256} \times \mathcal{L}(\mathit{Ev}_{\mathit{log}}) \times \mathcal{P}(\mathcal{A})$ with $b \in \mathbb{N}_{256}$ being the refund balance that is increased by account storage operations and will finally be paid to the transaction's beneficiary, $L \in \mathcal{L}(\mathit{Ev}_{\mathit{log}})$ being the sequence of log events that the bytecode execution invoked during execution and $S_\dagger \subseteq \mathcal{A}$ being the so-called suicide set – the set of account addresses that executed the SELFDESTRUCT command and therefore registered their account for deletion. The information held by the halting state is carried over to the calling state.

The state of a non-terminated internal transaction is described by a regular execution state of the form $(\mu, \iota, \sigma, \eta)$. The state is determined by the current global state σ of the system as well as the execution environment $\iota \in I$ that specifies the parameters of the current transaction (including inputs and the code to be executed), the local state $\mu \in M$ of the stack machine, and the transaction effects $\eta \in N$ collected during execution so far.

Execution Environment

The execution environment ι of an internal transaction specifies the static parameters of the transaction. It is a tuple of the form $(\mathit{actor}, \mathit{input}, \mathit{sender}, \mathit{value}, \mathit{code}) \in I = \mathcal{A} \times [\mathbb{B}^8] \times \mathcal{A} \times \mathbb{N}_{256} \times [\mathbb{B}^8]$ with the following components:

- $\mathit{actor} \in \mathcal{A}$ is the address of the account currently executing;
- $\mathit{input} \in [\mathbb{B}^8]$ is the data given as an input to the internal transaction;

- $sender \in \mathcal{A}$ is the address of the account that initiated the internal transaction;
- $value \in \mathbb{N}_{256}$ is the value transferred by the internal transaction;
- $code \in [\mathbb{B}^8]$ is the code currently executed.

This information is determined at the beginning of an internal transaction execution and it can be accessed, but not altered during the execution.

Machine State

The local machine state μ represents the state of the underlying state machine used for execution and is a tuple of the form (gas, pc, m, i, s) where

- $gas \in \mathbb{N}_{256}$ is the current amount of gas still available for execution;
- $pc \in \mathbb{N}_{256}$ is the current program counter;
- $m \in \mathbb{B}^{256} \to \mathbb{B}^8$ is a mapping from 256-bit words to bytes that represents the local memory;
- $i \in \mathbb{N}_{256}$ is the current number of active words in memory;
- $s \in \mathcal{L}(\mathbb{B}^{256})$ is the local 256-bit word stack of the stack machine.

The execution of each internal transaction starts in a fresh machine state, with an empty stack, memory initialized to all zeros, and program counter and active words in-memory set to zero. Only the gas is instantiated with the gas value available for the execution.

2.3.4 Small-Step Rules

In the following, we will present a selection of interesting small-step rules in order to illustrate the most important features of the semantics.

For demonstrating the overall design of the semantics, we start with the example of the arithmetic expression ADD performing addition of two values on the machine stack. Note that as the word size of the stack machine is 256, all arithmetic operations are performed modulo 2^{256}.

$$\frac{\iota.code\,[\mu.\mathsf{pc}] = \mathsf{ADD} \qquad \mu.\mathsf{s} = a :: b :: s \qquad \mu.\mathsf{gas} \geq 3 \qquad \mu' = \mu[\mathsf{s} \to (a+b) :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 3]}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\iota.code\,[\mu.\mathsf{pc}] = \mathsf{ADD} \qquad (|\mu.\mathsf{s}| < 2 \vee \mu.\mathsf{gas} < 3)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to \mathit{EXC} :: S}$$

We use a dot notation to access components of the different state parameters. We name the components with the variable names introduced for these components in the last section, written in sans-serif-style. In addition, we use the usual notation for updating components: $t[\mathsf{c} \to v]$

denotes that the component c of tuple t is updated with value v. For expressing incremental updates in a simpler way, we additionally use the notation $t[\mathsf{c} \mathrel{+}= v]$ to denote that the (numerical) component of c is incremented by v and similarly $t[\mathsf{c} \mathrel{-}= v]$ for decrementing a component c of t.

The execution of the arithmetic instruction ADD only performs local changes in the machine state, affecting the local stack, the program counter, and the gas budget. For deciding upon the correct instruction to execute, the currently executed code (that is part of the execution environment) is accessed at the position of the current program counter. The cost of an ADD instruction is constantly three units of gas that get subtracted from the gas budget in the machine state. As every other instruction, ADD can fail due to lacking gas or due to underflows on the machine stack. In this case, the exception state is entered, and the execution of the current internal transaction is terminated. For better readability, we use here the slightly sloppy $\vee$ notation for combining the two error cases in one inference rule.

A more interesting example of a semantic rule is the one of the CALL instruction that initiates an internal call transaction. In the case of calling, several corner cases need to be treated, which results in several inference rules for this case. Here, we only present one rule for illustrating the main functionality. More precisely, we present the case in that the account that should be called exists, the call stack limit of 1024 is not reached yet, and the account initiating the transaction has a sufficiently large balance for sending the specified amount of wei to the called account.

$$\frac{\begin{array}{c}\iota.\mathit{code}\,[\mu.\mathsf{pc}] = \mathsf{CALL} \qquad \mu.\mathsf{s} = g :: \mathit{to} :: \mathit{va} :: \mathit{io} :: \mathit{is} :: \mathit{oo} :: \mathit{os} :: s \\ \sigma(\mathit{to}) \neq \bot \qquad |A| + 1 < 1024 \qquad \sigma(\iota.\mathsf{actor}).\mathsf{b} \geq \mathit{va} \qquad \mathit{aw} = M\,(M\,(\mu.\mathsf{i}, \mathit{io}, \mathit{is}), \mathit{oo}, \mathit{os}) \\ c_{\mathit{call}} = C_{\mathit{gascap}}\,(\mathit{va}, 1, g, \mu.\mathsf{gas}) \qquad c = C_{\mathit{base}}\,(\mathit{va}, 1) + C_{\mathit{mem}}\,(\mu.\mathsf{i}, \mathit{aw}) + c_{\mathit{call}} \\ \mu.\mathsf{gas} \geq c \qquad \sigma' = \sigma\langle \mathit{to} \to \sigma(\mathit{to})[\mathsf{b} \mathrel{+}= \mathit{va}]\rangle\langle \iota.\mathsf{actor} \to \sigma(\iota.\mathsf{actor})[\mathsf{b} \mathrel{-}= \mathit{va}]\rangle \\ d = \mu.\mathsf{m}\,[\mathit{io}, \mathit{io} + \mathit{is} - 1] \qquad \mu' = (c_{\mathit{call}}, 0, \lambda x.\, 0, 0, \epsilon) \\ \iota' = \iota[\mathsf{sender} \to \iota.\mathsf{actor}][\mathsf{actor} \to \mathit{to}][\mathsf{value} \to \mathit{va}][\mathsf{input} \to d][\mathsf{code} \to \sigma(\mathit{to}).\mathsf{code}]\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to\; (\mu', \iota', \sigma', \eta) :: (\mu, \iota, \sigma, \eta) :: S}$$

For performing a call, the parameters to this call need to be specified on the machine stack. These are the amount of gas g that should be given as budget to the call, the recipient *to* of the call and the amount *va* of wei to be transferred with the call. In addition, the caller needs to specify the input data that should be given to the transaction and the place in memory where the return data of the call should be written after successful execution. To this end, the remaining arguments specify the offset and size of the memory fragment that input data should be read from (determined by *io* and *is*), and return data should be written to (determined by *oo* and *os*).

Calculating the cost in terms of gas for the execution is quite complicated in the case of CALL as it is influenced by several factors, including the arguments given to the call and the current machine state. First of all, the gas that should be given to the call (here denoted by c_{call}) needs to be determined. This value is not necessarily equal to the value g specified on the stack but also depends on the value *va* transferred by the call and the currently available gas. In addition, as the memory needs to be accessed for reading the input value and writing the return value, the number of active words in memory might be increased. This effect is captured by the memory extension function M. As accessing additional words in memory costs gas, this cost needs to

be taken into account in the overall cost. The costs resulting from an increase in the number of active words are calculated by the function C_{mem}. Finally, there is also a base cost charged for the call that depends on the value *va*. As the cost also depends on the specific case for calling that is considered, the cost calculation functions receive a flag (here 1) as arguments. These technical details are spelled out in Appendix A.2.

The call itself then has several effects: First, it transfers the balance from the executing account (*actor* in the execution environment) to the recipient (*to*). To this end, the global state is updated. Here we use a special notation for the functional update on the global state using $\langle\rangle$ instead of []. Second, for initializing the execution of the initiated internal transaction, a new regular execution state is placed on top of the execution stack. The internal transaction starts in a fresh machine state at program counter zero. This means that the initial memory is initialized to all zeros. Consequently, the number of active words in memory is zero as well, and the initial stack is empty. The gas budget given to the internal transaction is c_{call} calculated before. The transaction environment of the new call records the call parameters. This includes the sender that is the currently executing account *actor*, the new active account that is now the called account *to* as well as the value *va* sent, and the input data given to the call. To this end, the input data is extracted from the memory using the offset *io* and the size *is*. We use an interval notation here to denote that a part of the memory is extracted. Finally, the code in the execution environment of the new internal transaction is the code of the called account.

Note that the execution state of the caller stays completely unaffected at this stage of the execution. This is a conscious design decision in order to simplify the expression of security properties and to make the semantics more suitable to abstractions.

Besides CALL there are two different instructions for initiating internal call transactions that implement slight variations of the simple CALL instruction. These variations are called CALLCODE and DELEGATECALL, which both allow for executing another's account code in the context of the caller.

In the case of CALLCODE, the *actor* of the internal transaction is set to be the caller, which results in the callee's code modifying the caller's state when being executed. This behavior was intended for using library functionalities implemented in a separate library contract that, e.g., transfer money on behalf of the caller.

This idea is pushed even further in the DELEGATECALL instruction. This call type does not allow for transferring money and executes the callee's code not only in the caller's context but even preserves part of the execution environment of the previous call (in particular, the call value and the sender information). Intuitively, this instruction resembles adding the callee's code to the caller as an internal function so that calling it does not cause a new internal transaction (even though it formally does). By now, the usage of CALLCODE is deprecated in favor of DELEGATECALL.

Figure 2.2 summarizes the behavior of the different call instructions in EVM bytecode. The executed code of the respective account is highlighted in orange, while the accessible account state is depicted in green. The remaining internal transaction information (as specified in the execution environment) on the sender of the internal transaction and the transferred value are

marked in violet. In addition, the picture relates the corresponding changes to the small-step semantics: the execution of a call transaction adds a new execution state to the call stack while preserving the old one. The new global state σ' records the changes in the accounts' balances, while the new execution environment ι' determines the accessible account (by setting the actor of the internal transaction correspondingly), the code to be executed (by setting code) and further accessible transaction information as the sender, value and input (by setting sender, value and input respectively). In the case of CALLCODE and DELEGATECALL the global state in the new execution state stays unchanged due to the absence of money transfers.

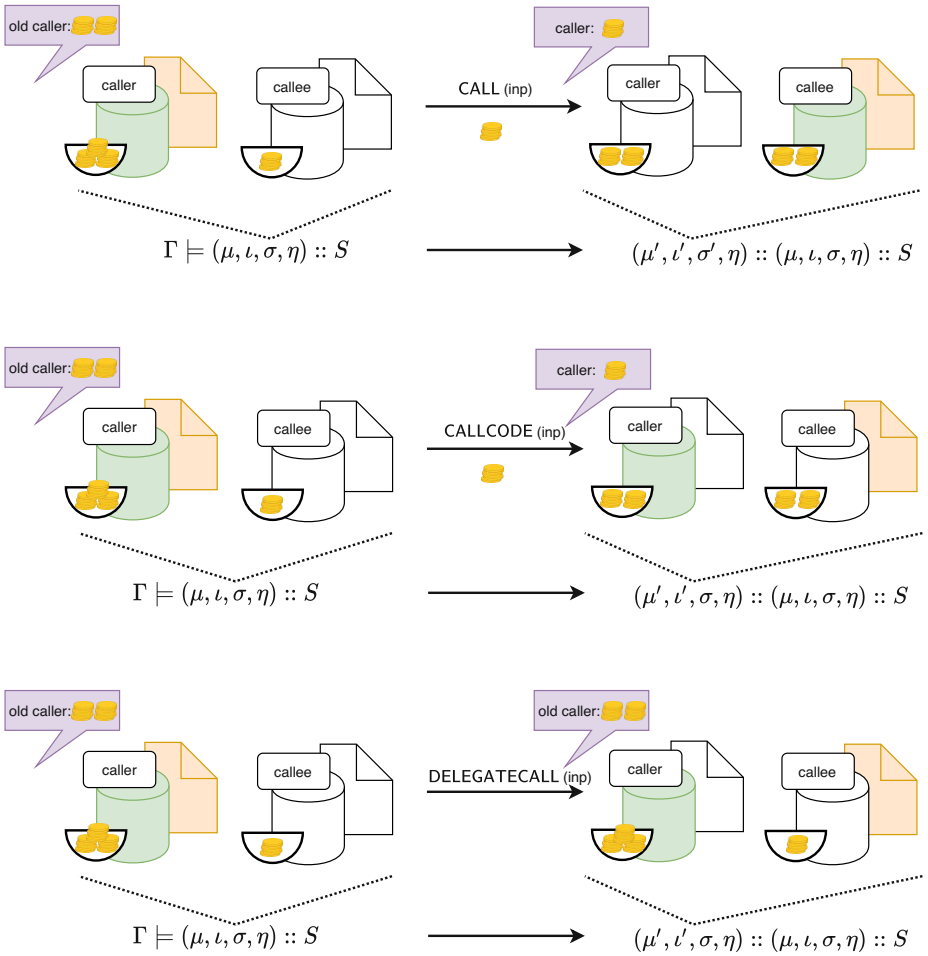

Figure 2.2: Illustration of of the semantics of different call types

Analogously to the instructions for initiating internal call transactions, there is also one instruction CREATE that allows for the creation of a new account. The semantics of this instruction is similar to the one of CALL, with the exception that a fresh account is created, which gets the specified

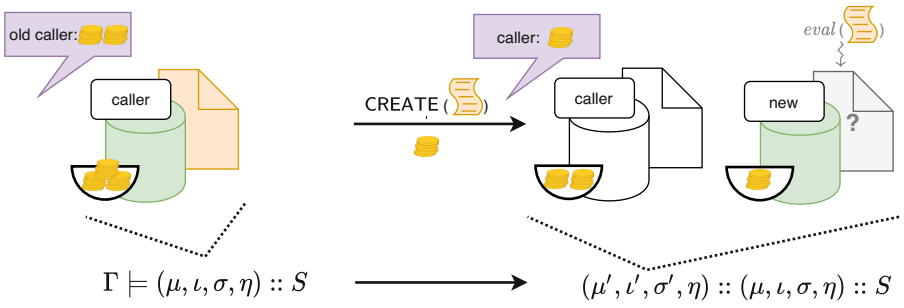

Figure 2.3: Illustration of the semantics of the CREATE instruction

transferred value, and that the input provided to this internal transaction, which is again specified in the local memory, is interpreted as the initialization code to be executed in order to produce the newly created account's code as output. In contrast to the call transaction, a create transaction does not await a return value but only an indication of success or failure.

Figure 2.3 depicts the semantics of the CREATE instruction in a similar fashion as it is done for the call instructions before. It is notable that the input to the CREATE instruction is interpreted as code and executed (therefore highlighted in orange) in the context of the newly created contract (highlighted in green). During this execution, the newly created contract does not have any contract code itself (therefore depicted in gray), but only after completing the internal transaction the return value of the transaction will be set as code for the freshly created contract.

For discussing how to return from an internal transaction, we show the rule for returning from a successful internal call transaction.

$$\frac{\begin{array}{c}\iota.code\,[\mu.\mathsf{pc}] = \mathsf{CALL} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \\ flag = \sigma(to) = \bot\,?\,0 : 1 \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(va, flag, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(va, flag) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu' = \mu[\mathsf{i} \to aw][\mathsf{s} \to 1 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c][\mathsf{m} \to \mu.\mathsf{m}[[oo, oo + s - 1] \to d]]\end{array}}{\Gamma \vDash HALT(\sigma', gas, d, \eta') :: (\mu, \iota, \sigma, \eta) :: S \;\to\; (\mu', \iota, \sigma', \eta') :: S}$$

Leaving the caller state unchanged at the point of calling has the negative side effect that the cost calculation needs to be redone at this point in order to determine the new gas value of the caller state. But besides this, the rule is straightforward: the program counter is incremented as usual, and the number of active words in memory is adjusted as memory accesses for reading the input and return data have been made. The gas is decreased, meaning that the overall amount of gas c allocated for the execution is subtracted. However, as this cost already includes the gas budget given to the internal transaction, the gas *gas* that is left after the execution is refunded again. In addition, the return data d is written to the local memory of the caller at the place specified by oo and os. Finally, the value one is written to the caller's stack in order to indicate the success of the internal call transaction. As the execution was successful, as indicated by the halting state, the global state and the transaction effects of the callee are adopted by the caller.

EVM bytecode offers several instructions for explicitly halting (internal) transaction execution. Besides the standard instructions STOP and RETURN, there is the SELFDESTRUCT instruction that is very particular to the blockchain setting. The STOP instruction causes regular halting of the internal transaction without returning data to the caller. In contrast, the RETURN instruction allows one to specify the memory fragment containing the return data that will be handed to the caller.

Finally, the SELFDESTRUCT instruction halts the execution and lists the currently executing account for later deletion. More precisely, this means that this account will be deleted when finalizing the external transaction, but its behavior during the ongoing small-step execution is not affected. Additionally, the whole balance of the deleted account is transferred to some beneficiary specified on the machine stack.

We show the small-step rules depicting the main functionality of SELFDESTRUCT. As for CALL, capturing the whole functionality of SELFDESTRUCT would require considering several corner cases. Here we consider the case where the beneficiary exists, the stack does not underflow and the available amount of gas is sufficient.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{SELFDESTRUCT} \\ \mu.\mathsf{s} = a_{ben} :: s \qquad a = a_{ben} \mod 2^{160} \qquad \sigma(a) \neq \bot \qquad \mu.\mathsf{gas} \geq 5000 \qquad g = \mu.\mathsf{gas} - 5000 \\ \sigma' = \sigma\langle a \to \sigma(a)[\mathsf{balance} \mathrel{+}= \sigma.(\iota.\mathsf{actor}).\mathsf{balance}]\rangle\langle \iota.\mathsf{actor} \to \sigma(\iota.\mathsf{actor})[\mathsf{balance} \to 0]\rangle \\ r = (\iota.\mathsf{actor} \in \Gamma.S_\dagger)\,?\,0 \,:\, 24000 \qquad \eta' = \eta[\mathsf{S}_\dagger \to \eta.\mathsf{S}_\dagger \cup \{\iota.\mathsf{actor}\}][\mathsf{balance} \mathrel{+}= r]\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to \mathit{HALT}(\sigma', g, \epsilon, \eta') :: S}$$

The SELFDESTRUCT command takes one argument a_{ben} from the stack specifying the address of the beneficiary that should get the balance of the account that is destructed. If all preconditions are satisfied, the balance of the executing account ($\iota.\mathsf{actor}$) is transferred to the beneficiary address, and the current internal transaction execution enters a halting state. Additionally, the transaction effects are extended by adding $\iota.\mathsf{actor}$ to the suicide set and by possibly increasing the refund balance. The refund balance is only increased in case that $\iota.\mathsf{actor}$ is not already scheduled for deletion. The halting state captures the global state σ after the money transfer, the remaining gas g after executing the SELFDESTRUCT and the updated transaction effects η'. As no return data is handed to the caller, the empty byte array ϵ is specified as return data in the halting state.

Note that SELFDESTRUCT deletes the currently executing account $\iota.\mathsf{actor}$ which is not necessarily the same account as the one owning the code $\iota.\mathsf{code}$. This might be due to the previous execution of DELEGATECALL or CALLCODE.

2.3.5 Transaction Execution

The outcome of an external transaction execution does not only consist of the result of the EVM bytecode execution. Before executing the bytecode, the transaction environment and the execution environment are determined from the transaction information and the block header. In the following, we assume $\mathcal{T}$ to denote the set of transactions. An (external) transaction $T \in \mathcal{T}$, similar to the internal transactions, specifies a gas limit, a recipient, and a value to be transferred.

In addition, it also contains the originator and the gas price that will be recorded in the transaction environment. Finally, it specifies an input to the transaction and the transaction type that can either be a call or a create transaction. The transaction type determines whether the input will be interpreted as input data to a call transaction or as an initialization code for a create transaction. In addition to the transaction of the environment initialization, some initial changes on the global state and validity checks are performed. For the sake of presentation we assume in the following a function $\mathit{initialize}\,(\cdot,\cdot,\cdot) \in \mathcal{T} \times \mathcal{H} \times \Sigma \to (\mathcal{T}_{env} \times \mathcal{S}) \cup \{\bot\}$ performing the initialization phase and returning a transaction environment and initial execution state in the case of a valid transaction and $\bot$ otherwise. Similarly, we assume a function $\mathit{finalize}\,(\cdot,\cdot,\cdot) \in T \times \mathcal{S} \times N \times \Sigma$ that given the final global state of the execution, the accumulated transaction effects and the transaction, computes the final effects on the global state. These include the deletion of the contracts from the suicide set and the payout to the beneficiary of the transaction.

Formally we can define the execution of a transaction $T \in \mathcal{T}$ in a block with header $H \in \mathcal{H}$ as follows:

$$\frac{(\Gamma, s) = \mathit{initialize}\,(T, H, \sigma) \qquad \Gamma \vDash s :: \epsilon \to^* s' :: \epsilon \qquad \mathit{final}\,(s') \qquad \sigma' = \mathit{finalize}\,(s', \eta', T)}{\sigma \xrightarrow{T,H} \sigma'}$$

where $\to^*$ denotes the reflexive and transitive closure of the small-step relation and the predicate $\mathit{final}\,(\cdot)$ characterizes a state that cannot be further reduced using the small-step relation. Note that to highlight here the overall workings of transaction execution, we omitted some details. In particular, the execution of an individual transaction cannot be considered in full isolation since also every block comes with a gas limit that the transactions consume in order of their execution. For the full formal details we refer to Appendix A.5.

2.3.6 Formalization in F*

We provide a formalization of a large fragment of our small-step semantics in the proof assistant F* [fst]. F* is an ML-dialect that is optimized for program verification and allows for performing manual proofs as well as automated proofs leveraging the power of SMT solvers.

Our formalization strictly follows the small-step semantics as presented in this chapter. The core functionality is implemented by the function `step` that describes how an execution stack evolves within one execution state. To this end, it has two possible outcomes: either it performs an execution step and returns the new call stack or – in the case that a final configuration is reached (which is a stack containing only one element that is either a halting or an exception state) – it reports the final state. In order to provide a total function for the step relation, we needed to introduce a third execution outcome that signalizes that a problem occurred due to an inconsistent state. When running the semantics from a valid initial configuration, this result, however, should never be produced. For running the semantics, the function `execution` is defined that subsequently performs execution steps using `step` until reaching the final state and reports it.

The current implementation encompasses approximately a thousand lines of code. Since F* code can be compiled into OCaml, we validate our semantics against the official EVM test suite [evm].

Our semantics passes 304 out of 624 tests, failing only in those involving any of the missing functionalities.

We make the formalization in F* publicly available [GMS18c] in order to facilitate the design of static analysis techniques for EVM bytecode as well as their soundness proofs.

2.3.7 Comparison with the Semantics by Luu et al. [LCO+16a]

The small-step semantics defined by Luu et al. [LCO+16a] encompasses only a variation of a subset of EVM bytecode instructions (called EtherLite) and assumes a heavily simplified execution configuration. The instructions covered span simple stack operations for pushing and popping values, conditional branches, binary operations, instructions for accessing and altering local memory and account storage, as well as as the ones for calling, returning, and destructing the account. Essential instructions as CREATE and those for accessing the transaction and block information are omitted. The authors represent a configuration as a tuple of a call stack of activation records and the global state. An activation record contains the code to be executed, the program counter, the local memory, and the machine stack. The global state is modeled as a mapping from addresses to accounts, with the latter consisting of code, balance, and persistent storage.

The overall abstraction contains a conceptual flaw, as not including the global state in the activation records of the call stack does not allow for modeling that, in the case of an exception in the execution of the callee, the global state is rolled back to the one of the caller at the point of calling. In addition, the model cannot be easily extended with further instructions – such as further call instructions or instructions accessing the environment – without major changes in the abstraction as a lot of information, e.g., the one captured in our small-step semantics in the transaction and the execution environment, are missing.

2.4 Security Definitions

In the following, we introduce the semantic characterization of the most significant security properties for smart contracts, motivating them with typical vulnerabilities recurring in the wild.

For selecting those properties, we inspected the classification of bugs performed in [LCO+16a] and [ABC17].

For the presented bugs, we synthesized the semantic security properties that were violated. In this process, we realized that some bugs share the same underlying property violation and that other bugs can not be captured by such generic properties – either because they are of a purely syntactic nature or because they constitute a derivation from the desired behavior that is particular to a specific contract.

Preliminary Notations Formally, we represent a contract as a tuple of the form $(a, code)$ where $a \in \mathcal{A}$ denotes the address of the contract and $code \in [\mathbb{B}]$ denotes the contract's code. We denote the set of contracts by $\mathcal{C}$ and assume functions $address\,(\cdot)$ and $code\,(\cdot)$ that extract the contract address and code respectively.

As we will argue about contracts being called in an arbitrary setting, we additionally introduce the notion of *reachable configuration*. Intuitively, a pair (Γ, S) of a transaction environment Γ and a call stack S is reachable if there exists a state s such that S, s are the result of *initialize* (T, H, σ), for some transaction T, block header H, a global state σ, and S is reachable from s.

Definition 1 (Reachable Configuration)**.** *The pair $(\Gamma, A) \in \mathcal{T}_{env} \times \mathcal{S}$ is a reachable configuration if for some transaction $T \in \mathcal{T}$, some block header $H \in \mathcal{H}$ and some global state $\sigma \in \mathcal{A} \to \mathbb{A}$ of the blockchain it holds that*

$$(\Gamma, s) = \mathit{initialize}\,(T, H, \sigma) \wedge \Gamma \vDash s :: \epsilon \;\to^{*} S$$

In order to give concise security definitions, we further introduce and assume throughout the chapter, an annotation to the small step semantics in order to highlight the contract c that is currently executed. Specifically, we let

$$\begin{aligned}\mathbb{S}_n := \{&\mathit{EXC}_c :: S_{plain},\ \mathit{HALT}(\sigma, gas, d, \eta)_c :: S_{plain},\ S_{plain}\\ &\mid \sigma \in \Sigma,\ gas \in \mathbb{N},\ d \in [\mathbb{B}^8],\ \eta \in N,\ S_{plain} \in \mathcal{L}((M \times I \times \Sigma \times N) \times \mathcal{C})\}\end{aligned}$$

Next, we introduce the notion of execution trace for smart contract execution. Intuitively, a trace is a sequence of actions. In our setting, the actions to be recorded are composed of an opcode, the address of the executing contract, and a sequence of arguments to the opcode. We denote the set of actions with *Act*. Accordingly, every small step produces a trace consisting of a single action. Again, we lift the resulting trace semantics to multiple execution steps that then produce sequences of actions $\pi \in \mathcal{L}(\mathit{Act})$. We only report the trace semantics definition for the CALL case here, referring to Appendix A.2 for further details.

$$\frac{\iota.\mathit{code}\,[\mu.\mathsf{pc}] = \mathsf{CALL}}{\mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \quad \cdots \quad \mu' = \cdots \quad \iota' = \cdots \quad \sigma' = \cdots}$$
$$\Gamma \vDash (\mu, \iota, \sigma)_c :: S \xrightarrow{\mathsf{CALL}_c(g,to,io,is,oo,os)} (\mu', \iota', \sigma')_{(to,\sigma(to).\mathsf{code})} :: (\mu, \iota, \sigma)_c :: S$$

We will write $\pi \downarrow_{\mathsf{calls}_c}$ to denote the projection of π to calls performed by contract c, i.e., actions of the form $\mathsf{CALL}_c(g, to, va, io, is, oo, os)$, $\mathsf{CREATE}_c(va, io, is)$, $\mathsf{CALLCODE}_c(g, to, va, io, is, oo, os)$, and $\mathsf{DELEGATECALL}_c(g, to, io, is, oo, os)$.

2.4.1 Call Integrity

Dependency on Attacker Code One of the most famous bugs of Ethereum's history is the so-called DAO bug that led to a loss of 60 million dollars in June 2016 [the16] and that we overviewed in Chapter 1. This bug is in the literature classified as reentrancy bug [ABC17, LCO+16a] as the affected contract was drained out of money by subsequently reentering it and performing transactions to the attacker on behalf of the contract. More generally, the problem of this contract

was that malicious code was able to affect the outgoing money flows of the contract. The cause of such bugs mostly roots in the developer's misunderstanding of the semantics of *Solidity*'s call primitives. In general, calling a contract can invoke two kinds of actions: Transferring Ether to the contract's account or Executing (parts of) a contract's code. In particular, the `call` construct invokes the called contract's fallback function when no particular function of the contract is specified (Section 2.2). Consequently, the developer may expect an atomic value transfer where potentially another contract's code is executed.

Call Integrity In order to protect from this class of bugs, it is crucial to secure the code against being reentered before regaining control over the control flow. From a security perspective, the fundamental problem is that the contract behavior depends on untrusted code, even though this was not intended by the developer. We capture this intuition through a hyperproperty, which we name *call integrity*. The idea is that no matter how the attacker can schedule c (call stacks S and S' in the definition), the calls of c (traces $\pi \downarrow_{\mathsf{calls}_c}$, $\pi' \downarrow_{\mathsf{calls}_c}$) cannot be controlled by the attacker, even if c hands over the control to the attacker.

Definition 2 (Call Integrity)**.** *A contract $c \in \mathcal{C}$ satisfies call integrity for a set of trusted addresses $\mathcal{A}_T \subseteq \mathcal{A}$ if for all reachable configurations $(\Gamma, s_c :: S), (\Gamma, {s'}_c :: S')$ with s, s' differing only in the code with address in $\mathcal{A}/\mathcal{A}_T$, it holds that for all t, t'*

$$\Gamma \vDash s_c :: S \xrightarrow{\pi}{}^{*} t_c :: S \ \wedge \ \mathit{final}\,(t_c) \ \wedge \ \Gamma \vDash {s'}_c :: S' \xrightarrow{\pi'}{}^{*} {t'}_c :: S' \ \wedge \ \mathit{final}\,({t'}_c) \implies \pi \downarrow_{\mathsf{calls}_c} = \pi' \downarrow_{\mathsf{calls}_c}$$

2.4.2 Proof Technique for Call Integrity

We now establish a proof technique for call integrity based on local properties that are arguably easier to verify and that we show to imply call integrity. As a first observation, we identify the different ways in which external contracts can influence the execution of a smart contract c and introduce corresponding security properties :

Code Dependency The contract c might access (information on) the code of the untrusted contract via the EXTCODECOPY or the EXTCODESIZE instructions and make his behavior depend on those values;

Effect Dependency The contract c might call the untrusted contract and might depend on its execution effects and return value;

Reentrancy The contract c might call the untrusted contract, with the latter influencing the behavior of the former by performing changes to the global state itself or "on behalf" of c by reentering it and thereby potentially decreasing the balance of c.

The first two of these properties can be seen as value dependencies and, therefore, can be formalized as hyperproperties. The first property says that the calls performed by a contract should not be affected by the effects on the execution state produced by adversarial contracts.

Technically, we consider a contract c calling an adversarial contract c' (captured as $\Gamma \vDash s_c :: S \rightarrow {s''}_{c'} :: s_c :: S$ in the premise), which we let terminate in two arbitrary states s', t': we require that c's continuation code performs the same calls in both states.

Definition 3 ($\mathcal{A}_T$-effect Independence)**.** *A contract $c \in \mathcal{C}$ is $\mathcal{A}_T$-effect independent of for a set of trusted addresses $\mathcal{A}_T \subseteq \mathcal{A}$ if for all reachable configurations $(\Gamma, s_c :: S)$ such that $\Gamma \vDash s_c :: S \rightarrow {s''}_{c'} :: s_c :: S$ for some s'' and address $(c') \in \mathcal{A}/\mathcal{A}_T$, it holds that for all final states s', t' whose global state might differ in all components but the code from the global state of s,*

$$\begin{aligned} &\Gamma_{init} \vDash {s'}_{c'} :: s_c :: S \xrightarrow{\pi}{}^{*} {s''}_c :: S \ \wedge \ final\,(s'') \\ \wedge\, &\Gamma_{init} \vDash {t'}_{c'} :: s_c :: S \xrightarrow{\pi'}{}^{*} {t''}_c :: S \wedge final\,(t'') \\ &\implies \pi \downarrow_{calls_c} = \pi' \downarrow_{calls_c} \end{aligned}$$

The second property says that the calls of a contract should not be affected by the code read from the blockchain (e.g., the code does not branch on code read from the blockchain). To this end we introduce the notation $\Gamma \vDash s :: S \xrightarrow[f]{\pi}{}^{*} s' :: S$ to denote that the local small-step execution of state s on stack S under Γ results in several steps in state s' producing trace π given that in the local execution steps of EXTCODECOPY and EXTCODESIZE, which are the operations used to access the code on the global state, the code returned by these functions is determined by the partial function $f \in \mathcal{A} \rightarrow\!\!\!\!\!\rightarrow [\mathbb{B}]$ as opposed to the global state. In other words, we consider in the premise a contract c reading two different codes from the blockchain and terminating in both runs (captured as $\Gamma \vDash s_c :: S \xrightarrow[f]{\pi}{}^{*} {s'}_c :: S$ and $\Gamma \vDash s_c :: S \xrightarrow[f']{\pi'}{}^{*} {s''}_c :: S$), and we require that c performs the same calls in both runs.

Definition 4 ($\mathcal{A}_T$-code Independence)**.** *A contract $c \in \mathcal{C}$ is $\mathcal{A}_T$-code independent for a set of trusted addresses $\mathcal{A}_T \subseteq \mathcal{A}$ if for all reachable configurations $(\Gamma, s_c :: S)$ it holds for all local code updates $f, f' \in \mathcal{A} \rightarrow\!\!\!\!\!\rightarrow [\mathbb{B}]$ on $\mathcal{A}/\mathcal{A}_T$ that*

$$\begin{aligned} \Gamma \vDash s_c :: S \xrightarrow[f]{\pi}{}^{*} {s'}_c :: S \ \wedge \ final\,(s') \ \wedge \ \Gamma \vDash s_c :: S \xrightarrow[f']{\pi'}{}^{*} {s''}_c :: S \ \wedge \ final\,(s'') \\ \implies \pi \downarrow_{calls_c} = \pi' \downarrow_{calls_c} \end{aligned}$$

Both these independence properties can be overapproximated by static analysis techniques based on program dependence graphs [HS09], as done by Joana to verify non-interference in *Java*™ [SGG+14]. The idea is to traverse the dependence graph in order to detect dependencies between the sensitive sources, in our case, the data controlled by the adversary and returned to the contract, and the observable sinks, in our case, the local contract calls.

The last property constitutes a safety property. Specifically, single-entrancy states that it cannot happen that when reentering the contract c another call is performed before returning (i.e., after reentrancy, which we capture in the call stack as two distinct states with the same running contract c, the call stack cannot further increase).

Definition 5 (Single-entrancy)**.** *A contract $c \in \mathcal{C}$ is single-entrant if for all reachable configurations $(\Gamma, s_c :: S)$, it holds for all s', s'', S' that*

$$\Gamma \vDash s_c :: S \rightarrow^* {s'}_c :: S' +\!\!+ s_c :: S$$
$$\implies \neg\exists s'' \in \mathcal{S}, c' \in \mathcal{C}.\, \Gamma \vDash {s'}_c :: S' +\!\!+ s_c :: S \rightarrow^* {s''}_{c'} :: {s'}_c :: S' +\!\!+ s_c :: S$$

The property expresses that after reentering a contract c (in state s'_c) while executing a call initiated by the very same contract, it is not possible anymore to perform another internal transaction (which would result in adding another element $s''_{c'}$ to the call stack). Note that the call stack records the sequence of calling states. Hence the suffix $s_c :: S$ indicates a pending call initiated by the execution s of contract c.

This safety property can be approximated by syntactic conditions, for instance, done in the Oyente analyzer [LCO+16a]. However, a sound approximation proved to be challenging as will be discussed in Chapter 3.

Finally, the next theorem proves the soundness of our proof technique, i.e., the two independence properties and the single-entrancy property together entail call integrity.

Theorem 1. *[Proof Technique for Call Integrity (Simplified)] Let $c \in \mathcal{C}$ be a contract and $\mathcal{A}_T \subseteq \mathcal{A}$ be a set of trusted addresses. If c is $\mathcal{A}_T$-code independent, c is $\mathcal{A}_T$-effect independent, and c is single-entrant then c provides call integrity for $\mathcal{A}_T$.*

Proof Sketch. Let $(\Gamma, s_c :: S), (\Gamma, {s'}_c :: S')$ be reachable configurations such that s, s' differ only in the code with address in $\mathcal{A}_C$. We now compare the two small-step runs of those configurations. Due to $\mathcal{A}_C$-code independence, the execution until the first call to an address $a \in \mathcal{A}_C$ produces the same partial trace until the call to a. Indeed, we can express the runs under different address mappings through the code update from the $\mathcal{A}_C$-code independence property, as long as no call to one of the updated addresses is performed. When the first call to $a \in \mathcal{A}_C$ is performed, we know due to single-entrancy that the following call cannot produce any partial execution trace for any of the runs as this would imply that contract c is reentered and a call out of the contract is performed. Due to $\mathcal{A}_C$-code independence and $\mathcal{A}_C$-effect independence, the traces after returning must coincide till the next call to an address in $\mathcal{A}_C$. This argument can be iteratively applied until reaching the final state of the execution of c.

For the sake of presentation, we slightly simplified the theorem here. For a full statement of the theorem as well as its detailed proof, we refer the reader to Appendix A.7.

2.4.3 Atomicity

Exception Handling

As discussed in section 2.2, the way exceptions are propagated varies with the way contracts are called. In particular, in the case of `call` and `send`, exceptions are not propagated, but a manual check for the successful completion of the called function's execution is required. This behavior reflects the way exceptions are reported during bytecode execution: Instead of propagating up

through the call stack, the callee reports the exception to the caller by writing zero to the stack. In the context of Ethereum, the issue of exception handling is particularly delicate as due to the gas restriction, it might always happen that a call fails because it ran out of gas. Intuitively, a user would expect a contract not to depend on the concrete gas value that is given to it, with the exception that a contract might always fail completely (and consequently does not perform any changes on the global state). Such behavior would prevent contracts from entering an inconsistent state as the one presented in the following excerpt of a simple banking contract:

```
1 contract SimpleBank{mapping(address => uint) balances;
2   function withdraw(){ msg.sender.send(balances[msg.sender]));
3     balances[msg.sender] = 0;}}
```

The contract keeps a record of the user balances and provides a function that allows a user to withdraw its own balance – which results in an update of the record. A developer might not expect that the `send` might fail, but as it is on the bytecode level represented by a CALL instruction, in addition to the Ether transfer, code might be executed that runs out of gas. As a consequence, the contract would end up in a state where the money was not transferred (as all effects of the call are reverted in case of an exception). However, the internal balance record of the contract was still updated; consequently, the money cannot be withdrawn by the owner anymore.

Inspired by such situations where an inconsistent state is entered by a contract due to mishandled gas exceptions, we introduce the notion of *atomicity* of a contract. Intuitively, atomicity requires that the effects of the execution on the global state do not depend on the amount of gas available – except when an exception is triggered, in which case the overall execution should have no effect at all. The last condition is captured by requiring that the final global state is the same as the initial one for at least one of the two executions (intuitively, the one causing the exception).

Definition 6. *A contract $c \in \mathcal{C}$ satisfies atomicity if for all reachable configurations (Γ, S') such that $\Gamma \vDash S' \rightarrow s_c :: S$, it holds for all gas values $g, g' \in \mathbb{N}_{256}$ that*

$$\begin{aligned} &\Gamma \vDash s_c[\mu.\mathit{gas} \rightarrow g] :: S \rightarrow^* {s'}_c :: S \ \wedge \mathit{final}\,(s') \\ \wedge\ &\Gamma \vDash s_c[\mu.\mathit{gas} \rightarrow g'] :: S \rightarrow^* {s''}_c :: S \ \wedge \mathit{final}\,(s'') \\ &\implies s'.\sigma = s''.\sigma \vee s.\sigma = s'.\sigma \vee s.\sigma = s''.\sigma \end{aligned}$$

2.4.4 Independence of Miner controlled Parameters

Another particularity of the distributed blockchain environment is that users, while performing transactions, cannot make assumptions on large parts of the context in which their transaction will be executed. A part of this is due to the asynchronous nature of the system: it can always be that another transaction that alters the context was performed first. Actually, the situation is even more delicate as transactions are not processed in a first-come-first-serve manner, but miners have a big influence on the execution context of transactions. They can decide upon the order of the transactions in a block (and also sneak their own transactions in first), and in addition, they can even control some parameters as the block timestamp within a certain range. Consequently, contracts whose (outgoing) money flows depend either on miner-controlled block information or on state information (as the state of their storage or their balance) that might be changed by other

transactions are prone to manipulations by miners. A typical example adduced in the literature is the use of block timestamps as a source of randomness [ABC17, LCO+16a]. In a classical lottery implementation that randomly pays out to one of the participants and uses the block timestamp as a source of randomness, a malicious miner can easily influence the result in his favor by selecting a beneficial timestamp.

We capture the absence of the miner's influence by two definitions, one saying that the outgoing Ether flows of a contract should not be influenced by components of the transaction environment that can be (within a certain range) set by miners and the other one saying that the Ether flows should not depend on those parts of the contract state that might have been influenced by previously executed transactions. The first definition rules out what is in the literature often described as timestamp dependency [ABC17, LCO+16a].

First, we define *independence of* (parts of) *the transaction environment*. To this end, we assume $\mathcal{C}_\Gamma$ to be the set of components of the transaction environment and write $\Gamma =_{/c_\Gamma} \Gamma'$ to denote that the transaction environments Γ, Γ' are equal up to component c_Γ.

Definition 7 (Independence of the Transaction Environment)**.** *A contract $c \in \mathcal{C}$ is independent of a subset $I \subseteq \mathcal{C}_\Gamma$ of components of the transaction environment if for all $c_\Gamma \in I$ and all reachable configurations $(\Gamma, s_c :: S)$ it holds for all Γ' that*

$$
\begin{aligned}
& c_\Gamma(\Gamma) \neq c_\Gamma(\Gamma') \wedge \Gamma =_{/c_\Gamma} \Gamma' \\
& \wedge\, \Gamma \vDash s_c :: S \xrightarrow{\pi}{}^{*} {s'}_c :: S \;\wedge\; \mathit{final}\,(s') \;\wedge\; \Gamma' \vDash s_c :: S \xrightarrow{\pi'}{}^{*} {s''}_c :: S \;\wedge\; \mathit{final}\,(s'') \\
& \implies \pi \downarrow_{\mathit{calls}_c} = \pi' \downarrow_{\mathit{calls}_c}
\end{aligned}
$$

Next, we define the notion of *independence of the account state*. Formally, we capture this property by requiring that the outgoing Ether flows of the contract under consideration should not be affected by those parameters of the contract that might have been changed by previous executions, which are the balance, the account's nonce, and the account's persistent storage.

Definition 8 (Independence of Mutable Account State)**.** *A contract $c \in \mathcal{C}$ is independent of the account state if for all reachable configurations $(\Gamma, s_c :: S), (\Gamma, s_c :: S')$ with s, s' differing only in the nonce, balance and storage for address (c), it holds that*

$$
\begin{aligned}
\Gamma \vDash s_c :: S \xrightarrow{\pi}{}^{*} {s'}_c :: S \;\wedge\; \mathit{final}\,({s'}_c) \;\wedge\; \Gamma \vDash s_c :: S' \xrightarrow{\pi'}{}^{*} {s''}_c :: S \;\wedge\; \mathit{final}\,({s''}_c) \\
\implies \pi \downarrow_{\mathit{calls}_c} = \pi' \downarrow_{\mathit{calls}_c}
\end{aligned}
$$

As the other independence properties, both these properties can be statically verified using program dependence graphs.

2.4.5 Classification of Bugs

The previously presented security definitions are motivated by the bugs that were observed in real Ethereum smart contracts and studied in [LCO+16a] and [ABC17]. Table 2.1 gives an overview of the bugs from the literature that are ruled out by our security properties.

Our security properties do not cover all bugs described by Atzei et al. [ABC17], as some of the bugs do not constitute violations of general security properties, i.e., properties that are not specific to the particular contract implementation. There are two classes of bugs that we do not consider: The first-class deals with the occurrence of unexpected exceptions (such as the Gasless Send and the Call stack Limit bug), and the second class encompasses bugs caused by the *Solidity* semantics deviating from the programmer's intuitions (such as the Keeping Secrets, Type Cast, and Exception Disorders bugs).

The first class of bugs encompasses runtime exceptions that are hard to predict for the developer and that are consequently not handled correctly. Of course, it would be possible to formalize the absence of those particular kinds of exceptions as simple reachability properties using the small-step semantics. Still, such properties would not give any insight into the security of a contract: the fact that a particular exception occurs can be unproblematic in the case that proper exception handling is in place. In general, the notion of correct exception handling depends highly on the specific contract's intended behavior. For the special case of out-of-gas exceptions, we could introduce the notion of atomicity in order to capture a generic goal of proper exception handling. But such a notion is not necessarily sufficient for characterizing reasonable ways of dealing with other kinds of runtime exceptions.

The second class of bugs are introduced on the *Solidity* level and are similarly hard to account for by using generic security properties. Even though these bugs might all originate from similar idiosyncrasies of the *Solidity* semantics, the impact of the bugs on the contract's semantics might deviate a lot. This might result in violations of the security properties discussed before, but also in violating the contract's functional correctness. Consequently, catching those bugs might require the introduction of contract-specific correctness properties.

Finally, Atzei et al. [ABC17] discuss the Ether Lost in Transfer bug. This bug is introduced by sending Ether to addresses that do not belong to any contract or user, so-called orphan addresses.

Table 2.1: Bugs from [LCO+16a] and [ABC17] ruled out by the security properties

Security Property	**Bug**
Call Integrity	Reentrancy [ABC17, LCO+16a] Call to the Unknown [ABC17]
Atomicity	Mishandled Exceptions [ABC17, LCO+16a]
Independence of Mutable Account State	Transaction Order Dependency [LCO+16a] Unpredictable State [ABC17]
Independence of Transaction Environment	Timestamp Dependancy [LCO+16a] Time Constraints [ABC17] Generating Randomness [ABC17]

We could easily formalize a reachability property stating that no valid contract execution should ever send Ether to such an address. We omit such a definition here as it is quite straightforward, and at the same time, it is not a property that directly affects the security of an individual contract: Sending Ether to such an orphan address might have negative impacts on the overall system as money is effectively lost. For the specific contract sending this money, this bug can be seen as a corner case of sending Ether to an unintended address which rather constitutes a correctness violation.

2.4.6 Discussion

To highlight the general and semantic nature of the introduced security definitions, we compare our definitions with the verification conditions used in Oyente [LCO+16a]. Our investigation shows that the verification conditions adopted in this tool are neither sound nor complete.

For detecting mishandled exceptions, it is checked whether each CALL instruction in the contract code is directly followed by the ISZERO instruction that checks whether the top element of the stack is zero. Unfortunately, Oyente (although stated in the paper) does not implement this check, so that we needed to manually inspect the bytecodes for determining the outcomes of the syntactic check. As shown in Figure 2.4a a check for the caller returning zero does not necessarily imply a proper exception handling and, therefore, atomicity of the contract. This excerpt of a simple banking contract that keeps track of the users' balances allows users to withdraw their balances using the function `withdraw` checks for the success of the performed call but still does not react accordingly. It only makes sure that the number of successes is updated consistently but does not perform the update on the user's balance record according to the call outcome.

On the other hand, not performing the desired check does not imply the absence of atomicity, as illustrated in Figure 2.4b. Writing the outcome in some variable before checking it satisfies the negative pattern, but correct exception handling is performed.

For detecting timestamp dependency, Oyente checks whether the contract has a symbolic execution path with the timestamp (that is represented as its own symbolic variable) being included in one of its constraints. This definition however, does not capture the case shown in Figure 2.4c.

This contract is clearly timestamp-dependent as whether or not the function `pay` pays out some money to the sender depends on the timestamp set when creating the contract. A malicious miner could consequently manipulate the block timestamp for a transaction that creates such a contract in a way that money is paid out and then subsequently query it for draining it out. This is, however, not captured by the characterization of the property in Oyente as they only capture the local execution paths of the contract.

On the other hand, using the block timestamp in path constraints does not imply a dependency as can easily be seen by the example in Figure 2.4d.

For the transaction order dependency and the reentrancy property, we were unfortunately not able to reconcile the property characterization provided in the paper with the implementation of Oyente.

```
1 contract SimpleBank{
2  mapping(address => uint) bal;
3  uint successes;
4   function withdraw(){
5    if (msg.sender.send(bal[msg.sender]))
6     {  successes++; }
7    bal[msg.sender] = 0;}}
```

2.4.a: Exception handling: False negative

```
1 contract SimpleBank{
2  mapping(address => uint) bal;
3   function withdraw(){
4    bool b =
5     msg.sender.send(bal[msg.sender]);
6     if (b) bal[msg.sender] = 0;}}
```

2.4.b: Exception handling: False positive

```
1 contract Test{
2  uint time = block.timestamp;
3   function pay (){
4    if (time % 2 == 1){
5     msg.sender.send(100);}}}
```

2.4.c: Timestamp dependency: False negative

```
1 contract Test {
2  function pay (){
3   if (block.timestamp % 2 == 1 ||
4    block.timestamp % 2 == 0){
5     msg.sender.send(100);}}}
```

2.4.d: Timestamp dependency: False positive

```
1 contract Fund{
2  mapping(address => uint) shares;
3   function withdraw(){
4    if (msg.sender.send(shares[msg.sender]))
5     shares[msg.sender] = 0;}}
```

2.4.e: Reentrancy: False negative

```
1 contract Bob{
2  bool sent = false;
3   function ping(address c){
4    if (!sent) {
5     sent = true;
6     c.call.value(2)();}}}
```

2.4.f: Reentrancy: False positive

For checking reentrancy, according to the paper, it should be checked whether the constraints on the path leading to a CALL instruction can still be satisfied after performing the updates on the path (e.g., changing the storage). If so, the contract is flagged as reentrant. According to our understanding, this approach should not flag contracts that correctly guard their calls as reentrant. Still, by the version of Oyente provided with the paper, the contract in Figure 2.4f is tagged as reentrant.

There exists an updated version of Oyente [LCO+] that is able to precisely tag this contract as not reentrant, but we could not find any concrete information on the criteria used for checking this property. Still, we found out that the underlying characterization can not be sufficient for detecting reentrancy as the contract in Figure 2.4e is classified not to exhibit a reentrancy vulnerability even though it should as the `send` command also executes the recipient's callback function (even though with limited gas). The example is taken from the Solidity documentation [sol19] where it is listed as a negative example. In addition to these soundness issues, the reentrancy detection suffers from issues similar to those of other analyzers, e.g., the ones of ZEUS [KGDS18a]. It does not consider that a contract might reinvoked several times (modifying the storage but not performing calls) before finally performing a problematic reentering call. In this way, reentrancy protection can be disabled without being detected. A more detailed discussion of this issue and other fallacies in the detection of reentrancies attack will be conducted in Chapter 3.

For transaction order dependency, Oyente should check whether execution traces exhibiting

different Ether flows exist. But it turned out that not even a simple example of a transaction-order-dependent contract can be detected by any of the versions of Oyente.

2.5 Conclusions

We presented the first complete small-step semantics of EVM bytecode and formalized a large fragment thereof in the F* proof assistant, successfully validating it against the official Ethereum test suite. We further defined for the first time a number of salient security properties for smart contracts, relying on a combination of hyper- and safety properties. Our framework is available to the academic community in order to facilitate future research on rigorous security analysis of smart contracts.

CHAPTER 3

Trends and Challenges in the Security Analysis of Ethereum Smart Contracts

Abstract

Ethereum smart contracts are distributed programs running on top of the Ethereum blockchain. Since program flaws can cause significant monetary losses and can hardly be fixed due to the immutable nature of the blockchain, there is a strong need of tools that assist users in developing secure smart contracts and in evaluating the security of existing smart contracts. We give a systematic overview of the different trends in developing such security-enhancing tools and particularly focus on such tools that are automated and at the same time give provable security guarantees. These tools are of special importance since they are accessible to a wide range of users while providing a high level of security. Designing such analyzers, however, proved to be challenging and error-prone. We review the existing approaches to automated, sound, static analysis of Ethereum smart contracts and highlight prevalent issues in state of the art.

This chapter presents a compilation of two invited contributions done in collaboration with Ilya Grishchenko, Markus Scherer, and Matteo Maffei, respectively. The first contribution was published under the title 'Foundations and Tools for the Static Analysis of Ethereum Smart Contracts' [GMS18a] at the 30th International Conference on Computer Aided Verification (CAV 2018). From this paper, I included the systematization of existing approaches to Ethereum smart contract analysis that I conducted. The second contribution was published under the title 'The Good, The Bad and The Ugly: Pitfalls and Best Practices in Automated Sound Static Analysis of Ethereum Smart Contracts' [SSM20] at the 9th International Symposium on Leveraging Applications of Formal Methods (ISoLA 2020). From this paper, I included the survey on challenges in the design of sound static analysis tools for Ethereum smart contracts and the discussion of soundness issues in existing analyzers. The accompanying appendix of this chapter contains detailed examples and discussions of the soundness issues in state of the art.

3.1 Introduction

Blockchain technologies are revolutionizing the distributed system landscape, providing an innovative solution to the consensus problem leveraging probabilistic guarantees and incentives. In particular, they allow for the secure execution of payments, and more in general, computations, among mutually distrustful parties. While some cryptocurrencies, like Bitcoin [Nak08] provide only a limited scripting language tailored to payments, others, like Ethereum [Woo14b], support a quasi Turing complete[1] smart contract language, allowing for advanced applications such as trading platforms [NGW17, MM17], elections [MSH17], permission management [CKY18, AEVL16], data management systems [PM18, Adh17], or auctions [HSLC17, GY18]. With the growing complexity of smart contracts, however, also the attack surface grows. This is particularly problematic as smart contracts control real money flows and hence constitute an attractive target for attackers. In addition, due to the immutable nature of blockchains, smart contracts cannot be modified once they are uploaded to the blockchain, which makes the effects of security vulnerabilities permanent. This is not only a theoretical threat but a practical problem, as demonstrated by infamous hacks, such as the DAO hack [the16] or the Parity hacks [par17a, par17b] which caused losses of several millions of dollars. This state of affairs calls for the development of tools that assist the users of the Ethereum system. This includes tools for developers that help them to design secure smart contracts and to verify their contracts before uploading them to the blockchain, but also tools for users interacting with existing smart contracts, who need tool assistance to assess whether or not those contracts (which are published in human unreadable bytecode format on the blockchain) are fraudulent.

In this chapter, we systematize different approaches to developing tool support for the users of Ethereum smart contracts. We lay a special focus on tools that give provable guarantees on smart contract security while at the same time being automated and hence accessible to a wide range of users. For a proper classification of these tools, we first overview the different verification approaches in the realm of Ethereum smart contracts and afterward discuss in detail those tools that focus on automated and sound static analysis of Ethereum smart contracts. We critically reflect the methodology and soundness claims of these tools and use the observed shortcomings to characterize the particular challenges in the sound and static analysis of Ethereum smart contracts.

The remainder of the chapter is structured as follows: Section 3.2 surveys the recent developments in the design of tools for enhancing Ethereum smart contract security and the open challenges in this domain. Section 3.3 overviews the existing automated static analyzers for Ethereum smart contracts that come with soundness claims. Section 3.4 works out the particular challenges in the design of sound static analysis tools for Ethereum smart contracts taking as an example the hurdles that the prior works stumbled upon.

[1] Supporting a Turing-complete instruction set, Ethereum enforces termination by bounding the number of computation steps based on a prespecified resource limit.

3.2 Trends in Security-enhancing Tools for Ethereum Smart Contracts

In the following, we give systematization of the approaches are taken so far in the direction of securing (Ethereum) smart contracts. This systematization does not aim at giving an exhaustive overview of all tools developed in this domain but wants to characterize the different trends towards enhancing Ethereum smart contract security in the scientific community, illustrated by the pioneering works in the corresponding categories. We distinguish between verification approaches and design approaches. According to our terminology, the goal of verification approaches is to check smart contracts written in existing languages (such as *Solidity*) for their compliance with a security policy or specification. In contrast, design approaches aim at facilitating the creation of secure smart contracts by providing frameworks for their development: These approaches encompass new languages which are more amenable to verification, provide a clear and simple semantics that is understandable by smart contract developers or allow for direct encoding of desired security policies. In addition, we count works that aim at providing design patterns for secure smart contracts to this category.

3.2.1 Verification

In the field of smart contract verification, we categorize the existing approaches along the following dimensions: target language (bytecode vs. high-level language), point of verification (static vs. dynamic analysis methods), provided guarantees (bug-finding vs. formal soundness guarantees), checked properties (generic contract properties vs. contract specific properties), degree of automation (automated verification vs. assisted analysis vs. manual inspection). From the current spectrum of analysis tools, we can find solutions in the following clusters:

Static Analysis Tools for Automated Bug-finding

The bug-finding tool Oyente [LCO+16b] (published in 2016) pioneered the (automatic) static analysis of Ethereum smart contracts. This work highlighted, for the first time, generic types of bugs that typically affect smart contracts and proposed a tool based on symbolic execution for the detection of contracts vulnerable to these bugs. A particularly compelling feature of Oyente is that it is a push-button tool that does not expect any interaction or a deeper knowledge of the contract semantics from the user. Oyente supports a variety of predefined security properties, such as transaction order dependency, time-stamp dependency, and reentrancy, that can be checked automatically. However, Oyente does not provide any guarantees on the reported results, being neither sound (absence of false negatives) nor complete (absence of false positives) and, thereby, yielding only a heuristic indication of contract security. This is on the one hand due to the simplified semantics that serves as foundation of the analysis, as discussed in Section 2.3.7. On the other hand, as detailed out in Section 2.4.6, the security properties are rather syntactic or pattern-based and are lacking a semantic characterization. Recently, Zhou et al. proposed the static analysis tool SASC [ZHP+18] that extends Oyente by additional patterns and provides a visualization of detected risks in the topology diagram of the original *Solidity* code.

Majan [NKS+18] extends the approach taken in Oyente to trace properties that consider multiple invocations of one smart contract. As Oyente, it relies on symbolic execution that follows a simplified version of the semantics used in Oyente and uses a pattern-based approach for defining the concrete properties to be checked. The tool covers safety properties (such as prodigality and suicidality) and liveness properties (greediness). As for Oyente, the authors do not make any security claims but consider their tool a 'bug-catching approach'.

Aside from works that build on the symbolic execution approach of Oyente, a broad variety of other automatic analysis tools have emerged in the last years. These tools base on different analysis techniques, such as the usage of high-performance datalog engines (like *Soufflé* [JSS16]) to reason about information and data flow dependencies in smart contracts [LGTS, BGL+20], the usage of bounded model checkers [FAH20, WZS19], or of existing verification tools for other programming languages [ACG+19]. Further, they target different forms of vulnerabilities. Next to tools that allow for encoding arbitrary properties out of certain property classes, such as [BGL+20] which presents a generic information flow framework, there exists a variety of other bug-finding tools targeting particular bug classes. Examples are the tool Osiris [TS+18] which focuses on the detection of integer bugs, or the tool NPChecker [WZS19] which focuses on the detection of non-deterministic payment bugs. We omit a more detailed discussion of these tools and refer the interested reader to recent surveys [DAS19, LL19] for more details.

Another approach towards bug finding in Ethereum smart contracts is taken by works that apply testing, fuzzing, or machine learning techniques to smart contracts [LLC+18, LWX+19, WSX+20, AMJC20]

Static Analysis Tools for Automated Verification of Generic Properties

In contrast to the aforementioned class of tools, this line of work focuses on providing formal guarantees for the analysis results while still performing an automated verification to ensure usability and broad adaption. Despite four years of intense research, however, only four works on such sound and fully automatic static analysis of Ethereum smart contracts have been published. Upon close investigation, it turns out that all of these works exhibit shortcomings that ultimately undermine the security guarantees that they aim to provide. We defer the detailed discussion of these tools to Section 3.3.

Frameworks for Semi-automated Proofs for Contract Specific Properties

Hirai [Hir17b] formalizes the EVM semantics in the proof assistant Isabelle/HOL and uses it for manually proving safety properties for concrete contracts. This semantics, however, constitutes a sound over-approximation of the original semantics [Woo14b]. Building on top of this work, Amani et al. propose a sound program logic for EVM bytecode based on separation logics [ABBS18]. This logic allows for semi-automatically reasoning about correctness properties of EVM bytecode using the proof assistant Isabelle/HOL.

Hildebrandt et al. [HSR+18] define the EVM semantics in the $\mathbb{K}$ framework [Rc10] – a language-independent verification framework based on reachability logics. The authors leverage the power of the $\mathbb{K}$ framework in order to automatically derive analysis tools for the specified semantics,

presenting as an example a gas analysis tool, a semantic debugger, and a program verifier based on reachability logics. The derived program verifier still requires the user to manually specify loop invariants on the bytecode level.

Bhargavan et al. [BDLF+16a] introduce a framework to analyze Ethereum contracts by a translation into F*, a functional programming language aimed at program verification and equipped with an interactive proof assistant. The translation supports only a fragment of the EVM bytecode and does not come with a justifying semantic argument.

Dynamic Monitoring for Predefined Security Properties

Grossman et al. [GAGG+17] propose the notion of effectively callback-free executions and identify the absence of this property in smart contract executions as the source of common bugs such as reentrancy. They propose an efficient online algorithm for discovering executions violating effectively callback freeness. Implementing a corresponding monitor in the EVM would guarantee the absence of the potentially dangerous smart contract executions but is not compatible with the current Ethereum version and would require a hard fork.

A dynamic monitoring solution compatible with Ethereum is offered by the tool DappGuard [CLL]. The tool actively monitors the incoming transactions to a smart contract and leverages the tool Oyente [LCO+16b], an own analysis engine and a simulation of the transaction on the testnet for judging whether the incoming transaction might cause a (generic) security violation (such as transaction order dependency). If a transaction is considered harmful, a counter transaction (killing the contract or performing some other fixes) is made. The authors claim that this transaction will be mined with high probability before the problematic one. Due to this uncertainty and the bug-finding tools used for the evaluation of incoming transactions, this approach does not provide any guarantees.

Recently, also other approaches to runtime verification of Ethereum smart contracts are explored, which either rely on inlining monitors into smart contracts [AEP18] or explore possibilities to enhance Ethereum's virtual machine to support monitoring [Ell20].

3.2.2 Design

The research on secure smart contract design focuses on the following four areas: high-level programming languages, intermediate languages (for verification), security patterns for existing languages, and visual tools for designing smart contracts.

High-level Languages

One line of research on high-level smart contract languages concentrates on the facilitation of secure smart contract design by limiting the language expressiveness and enforcing strong static typing discipline. Simplicity [O'C17] is a typed functional programming language for smart contracts that disallows loops and recursion. It is a general-purpose language for smart contracts and not tailored to the Ethereum setting. Simplicity comes with a denotional semantics specified in Coq that allows for formally reasoning about Simplicity contracts. As there is no (verified)

compiler to EVM bytecode so far, such results do not carry over to Etherum smart contracts. In the same realm, Pettersson and Edström [PE], propose a library for the programming language Idris that allows for the development of secure smart contracts using dependent and polymorphic types. They extend the existing Idris compiler with a generator for Serpent code (a Python-like high-level language for Ethereum smart contracts). This compiler is a proof of concept and fails in compiling more advanced contracts (as it cannot handle recursion). In preliminary work, Coblenz et al. [Cob17] propose Obsidian, an object-oriented programming language that pursues the goal of preventing common bugs in smart contracts such as reentrancy. To this end, Obsidian makes states explicit and uses a linear type system for quantities of money.

Another line of research focuses on designing languages that allow for encoding security policies that are dynamically enforced at runtime. The first step in this direction is sketched in the preliminary work on Flint [SED], a type-safe, capabilities-secure, contract-oriented programming language for smart contracts that is compiled to EVM bytecode. Flint allows for defining caller capabilities restricting access to security-sensitive functions. These capabilities shall be enforced by the EVM bytecode created during compilation.

In addition to these approaches from academia, the Ethereum foundation currently develops the high-level languages Viper [vip] and Bamboo [bam]. Furthermore, the *Solidity* compiler used to support a limited export functionality to the intermediate language WhyML [why] allowing for a pre-/postcondition style reasoning on *Solidity* code by leveraging the deductive program verification platform Why3 [FP13]. In more recent efforts [MOA+20] this approach was replaced by an extension of the *Solidity* compiler to directly model relevant parts of the *Solidity* smart contract semantics as Constrained Horn clauses to reason about them using SMT solvers.

Intermediate Languages

The intermediate language Scilla [SKH18] comes with a semantics formalized in the proof assistant Coq and therefore allows for a mechanized verification of Scilla contracts. In addition, Scilla makes some interesting design choices that might inspire the development of future high level languages for smart contracts: Scilla provides a strict separation not only between computation and communication but also between pure and effectful computations. Also, the analysis tool Slither [FGG19] makes use of a custom intermediate representation to reason about smart contracts. Further, the intermediate languages IELE [KGM+19] and Elle [ALV] were introduced.

Security Patterns

Wöhrer [WZ18] describes programming patterns in *Solidity* that should be adopted by smart contract programmers for avoiding common bugs. These patterns encompass best coding practices such as performing calls at the end of a function, but also off-the-shelf solutions for common security bugs such as locking a contract for avoiding reentrancy or the integration of a mechanism that allows the contract owner to disable sensitive functionalities in the case of a bug.

Tools

Mavridou and Laszka [ML18] introduce a framework for designing smart contracts in terms of finite state machines. They provide a tool with a graphical editor for defining contract specifications as automata and give a translation of the constructed finite state machines to *Solidity*. In addition, they present some security extensions and patterns that can be used as off-the-shelf solutions for preventing reentrancy and implementing common security challenges such as time constraints and authorization. However, the approach lacks formal foundations as neither the correctness of the translation is proven correct nor are the security patterns shown to meet the desired security goals.

3.2.3 Open Challenges

Even though the previous section highlights the wide range of steps taken towards the analysis of Ethereum smart contracts, there are still a lot of open challenges left.

Provably sound automated analysis tools for EVM bytecode

For ensuring the security of smart contracts whose source code is not public or that got manually optimized on the bytecode level, providing tools that operate on bytecode is crucial. Furthermore, due to contracts being immutable once published on the blockchain, the reliability of analysis results is of outstanding importance. For these reasons, developing provably sound analysis tools for different classes of properties relevant to smart contract security (such as reachability properties or value dependency properties) constitutes an immediate challenge in the realm of smart contract verification.

Up to now, all automated tools that aimed at providing provable guarantees failed to live up to these soundness claims. We will review these tools and the concrete difficulties that they encounter in Section 3.4 and finally show in Chapter 4 how we managed to overcome these issues with the sound static analyzer *eThor*.

Secure Compilation of High-level Languages

Even though there are several proposals made for new high-level languages that facilitate the design of secure smart contracts and that are more amenable to verification, none of them comes so far with a verified compiler to EVM bytecode. This particularly applies to the compilation from *Solidity*, which is still lacking a complete formal semantics, as described in a recent survey [Ari19]. However, such a secure compilation is the requirement for the results shown on high-level language programs to carry over to the actual smart contracts published on the blockchain.

Specification Languages for Smart Contracts

So far, most approaches to verifying contract-specific properties focus on either ad-hoc specifications in the used verification framework [Hir17b, HSR+18, BDLF+16a, ABBS18] or the insertion of assertions into existing contract code [why]. For leveraging the power of existing

model checking techniques for program verification, the design of a general-purpose specification language for Ethereum smart contracts would be needed.

Study of Security Policies

There has been no fundamental research made so far on the classes of security policies that might be interesting to enforce in the setting of smart contracts. In particular, it would be compelling to characterize the class of security policies that can be enforced by smart contracts within the existing EVM.

Compositional Reasoning about Smart Contracts

Most research on smart contract verification focuses on reasoning about individual contracts or, at most, a bunch of contracts whose bytecode is fully available. Even though there has been work observing the similarities between smart contracts and concurrent programs [SH17], there has been no rigorous study on compositional reasoning for smart contracts so far, even though first steps in this direction have been explored [QHC+18].

3.3 State of the Art in Automated Sound Static Analysis of Ethereum Smart Contracts

In the following, we focus on the *automated and sound* static analysis of Ethereum smart contracts. So far, there have been works on four static analyzers published that come with (explicit or implicit) soundness claims: the dependency analysis tool Securify [TDDC+18] for EVM bytecode, the static analyzer ZEUS [KGDS18b] for *Solidity*, the syntax-guided *Solidity* analyzer NeuCheck [LWZ+19], and the bytecode-based reachability analysis tool EtherTrust [GMS18a]. By implicit soundness claim, we mean that the tool claims that a positive analysis result guarantees the contract's security (i.e., absence of false negatives with respect to a specific security property). While Securify, ZEUS, and EtherTrust implement semantic-based analysis approaches, NeuCheck is purely syntax-driven.

Securify supports data and control flow analysis on the EVM bytecode level. To this end, it reconstructs the control-flow graph (CFG) from the contract bytecode and transforms it into SSA-form. Based on this structured format, it models immediate data and control flow dependencies using logical predicates and establishes datalog-style logical rules for deriving transitive dependencies, which are automatically computed using the enhanced datalog engine *Soufflé* [JSS16]. For checking different security properties, Securify specifies patterns based on the derived predicates, which shall be sufficient for either proving a property (compliance patterns) or for showing a property to be broken (violation patterns).

ZEUS analyzes *Solidity* contracts by first translating them into the intermediate language LLVM bitcode and then using off-the-shelf model checkers to verify different security properties. In the course of the translation, ZEUS uses another intermediate layer for the *Solidity* language, which introduces abstractions and that allows for the insertion of assumptions and assertions into

the program code, which express security requirements on the contract. The security properties supported by ZEUS are translated to such assertion checks, possibly in conjunction with additional property-specific contract transformations.

NeuCheck analyzes *Solidity* contracts by pattern matching on the contract syntax graph. To this end, it translates *Solidity* source code into an XML parse tree. Security properties are expressed as patterns on this parse tree and are matched by custom algorithms traversing the tree.

EtherTrust implements a reachability analysis on EVM bytecode by abstracting the bytecode execution semantics into (particular) logical implications, so-called Horn clauses, over logical predicates representing the execution state of a contract. Security properties are expressed as reachability queries on logical predicates and solved by the SMT solver *z3* [DMB08]. EtherTrust was the first prototype that later evolved into the *eThor* analyzer, which we will discuss in Chapter 4.

All presented tools focus on generic (contract-independent) security properties for smart contracts. However, the underlying architectures allow for extending the frameworks with further properties. For the soundness considerations in this chapter, we put the focus on the abstractions of generic security properties.

3.4 Challenges in Sound Smart Contract Verification

EVM bytecode exposes several domain-specific subtleties that turn out to be challenging for static analysis. Furthermore, characterizing relevant generic security properties for smart contracts is highly non-trivial and subject to ongoing research. We will examine both of these problems in the following.

3.4.1 Analysis Design

We summarize below the main challenges that arise when designing a performant and still sound analysis for Ethereum smart contracts:

- *Dynamic jump destinations:* Jump destinations are statically unknown and computed during execution. They might be influenced by the blockchain environment as well as the contract state. As a consequence, the control flow graph of a contract is not necessarily determinable at analysis time.
- *Mismatch between memory and stack layout:* The EVM has a (stack) word size of 256 bits while the memory (heap) is fragmented into bytes and addressed accordingly. Loading words from memory to the stack, and conversely, writing stack values to memory, requires (potentially costly) conversions between these two value formats.
- *Exception propagation and global state revocation:* If an internal transaction (as, e.g., initiated by a CALL) fails, all effects of this transaction, including those on the global state (e.g., writes to global storage), are reverted. However, such a failure is not propagated to the callee, who can

continue execution in the original global state. Modeling calls must thus save the state before calling in order to account for global state revocation.

- *Native support for low-level cryptography:* The EVM supports a designated SHA3 instruction to compute the hash of some memory fraction. As a consequence, hashing finds broad adaption in Ethereum smart contracts, and the *Solidity* compiler bases its storage layout on a hash-based allocation scheme.
- *Dynamic calls:* The recipient of an (inter-contract) call is specified on the stack and hence subject to prior computation. Consequently, the recipient is not necessarily derivable at analysis time, resulting in uncertainty about the behavior of the callee and the resulting effects on the environment.
- *Dynamic code creation:* Ethereum supports the generation of new smart contracts during transaction execution: A smart contract can deploy another one at runtime. To do so, the creating smart contract reads the deployment code for the new contract from the heap. The newly created contract may hence be subject to prior computation and even to the state of the blockchain.

In order to effectively tackle these challenges, several contributions of independent interest are required, such as domain-specific abstractions (e.g., suitable over-approximations of unknown environment behavior); the preprocessing of the contract to reconstruct its control flow or call graph; (easily checkable) assumptions that restrict the analysis scope (e.g., restriction to some language fragment); and optimizations or simplifications in intermediate processing steps (e.g, contract transformations to intermediate representations). Altogether, these analysis steps enlarge the semantic gap between the original contract semantics and the analysis, making it harder to reliably ensure the soundness of the latter. In the following, we will review in more detail the tension between soundness and performance of the analysis, and how past works stumbled in this minefield.

Soundness

Ensuring the soundness of the analysis requires a rigorous specification of the semantics of EVM bytecode. The original semantics was written in the Yellow Paper [Woo14b]. This paper however, from the beginning exhibited flaws(e.g., those pointed out in Chapter 2 as well as in other works [Hir17b, HSR+18]) and underspecified several aspects of bytecode execution. The ultimate truth of smart contract semantics could therefore only be extracted from the client implementations provided by the Ethereum foundation. In the course of time, in addition to the semantics presented in Chapter 2, several formal specifications of the EVM semantics have been proposed by the scientific community [HSR+18, Hir17b], leading to the Yellow paper being replaced by an executable semantics in the $\mathbb{K}$ framework [HSR+18][2].

For the high-level language *Solidity*, despite first efforts within the scientific community [CDPZ19, Zak18, BGM19, YL18, JKL+18], there exists at present no full and generally accepted formal

[2]Also called the Jello paper: `https://jellopaper.org`

```
1 contract Test {
2   bool test = false;
3   function flipper () { if (msg.sender != 0){flip();} }
4   function flip () internal {test = !test;} }
```

Figure 3.1: Simple contract highlighting an unsoundness in Securify's dependency analysis.

semantics. Consequently, the semantics of *Solidity* is only defined by its compilation to EVM bytecode. Since the compiler is subject to constant changes, *Solidity* constitutes a moving target.

The complexity and uncertainty about the concrete semantics made most works build on ad-hoc simplified versions of the semantics, which do not cover all language features and disregard essential aspects of the EVM's execution model.

ZEUS [KGDS18b], for instance, defines an intermediate goto language for capturing the core of *Solidity*. The semantics of this language, however, is inspired by the (ad-hoc) semantic modeling used in Oyente [LCO+16b], inheriting an essential flaw concerning global state revocation: In case that an internal transaction returns with an exception, the effects on the global state are not reverted as they would be in real EVM (and *Solidity*) executions. Since the translation to the intermediate language is part of the analysis pipeline of [KGDS18b], such a semantic flaw compromises the soundness of the whole analysis. A more detailed discussion is provided in Appendix B.1.5.

Also Securify [TDDC+18] introduces an ad-hoc formalism for EVM bytecode semantics. This is not, however, related to the dependency predicates used for the analysis but just serves for expressing security properties. It is hence unclear to which extent the dependency predicates faithfully reflect the control flow and value dependencies induced by the EVM semantics. Assessing the correctness of this approach is difficult since no full logical specification of the dependency analysis is provided[3]. Indeed we found empirical indication for the unsoundness of the dependency analysis in the presence of complicated control flow. Consider the example contract depicted in Figure 3.1. The depicted contract `Test` has a global boolean field `test` which will be saved in the persistent storage of the contract and hence constitutes the contract state. The public function `flipper()` allows every account but the one with address 0 to flip the value of the `test` field: For checking the restriction on the calling account, the `flipper()` function accesses the address of the caller using *Solidity*'s `msg.sender` construct. For writing the `test` field, the internal function `flip()` is called. Internal functions are not exposed to the public but are only accessible by the contract itself, and calls to such functions are compiled to local jumps. The use of internal functions consequently substantially complicates the control flow of a contract.

We identified a correctness issue that affects both the soundness and completeness of Securify. This incorrectness becomes evident in the violation pattern that checks for unrestricted writes. An unrestricted write is a write access to the global storage that any caller can perform. The violation

[3] Only an excerpt is presented in [TDDC+18], and the public implementation at `https://github.com/eth-sri/securify` intermingles specification and implementation.

pattern states that such an unrestricted write is guaranteed to happen if there is a SSTORE instruction whose reachability does not depend on the caller of the contract. This pattern should not be matched by the `Test` contract since the only write access to the `Test`'s sole variable `test` in function `flip()` is only reachable via the function `flipper` where it is conditioned on the caller (`msg.sender`). Hence not every contract can write the `test` variable, but the write access depends on the caller. Still, Securify reports this contract to match the violation pattern, consequently proving the wrong statement that there is no dependency between writing the `test` field and the account calling the contract. Note that even though showing up in a violation pattern (hence technically producing false positives), the underlying issue also affects the soundness of the core dependency analysis[4]. Securify specifies a *may dependency* relation to capture (potential) abstract dependencies between program locations. For correctly (i.e., soundly) abstracting the dependencies in real programs, the absence of a may dependency should imply corresponding independence in the real program. Since the may dependency relation is used in both compliance and violation patterns, without such a guarantee Securify can be neither sound nor complete. The example refutes this guarantee and thereby illustrates the importance of providing clear formal correctness (i.e., soundness) statements for the analysis.

These two examples show how the missing semantic foundations of the presented analysis approaches can lead to soundness issues in the core analysis design itself. These problems are further aggravated once additional stages are added to the analysis pipeline for increasing performance since such additional stages are often not part of the correctness considerations.

Performance

For performance reasons, it is often unavoidable to leverage well-established and optimized analysis frameworks or solvers. This leaves analysis designers with the challenge to transform their analysis problem into a format that is expressible and efficiently solvable within the targeted framework while preserving the semantics of the original problem.

ZEUS [KGDS18b] makes use of off-the-shelf model checkers for LLVM bitcode and hence requires a transformation of smart contracts to LLVM bitcode. The authors describe this step to be a 'faithful expression-to-expression translation' that is semantics preserving but omit a proof for this property. The paper itself later contradicts this statement: The authors report on LLVM's optimizer impacting the desired semantics. This indicates that the semantics of the LLVM bitcode translation does not coincide with the one of the intermediate language, since it would otherwise not be influenced by (semantics-preserving) optimizations.

The Securify tool [TDDC+18] makes use of several preprocessing steps in order to make EVM bytecode amenable to dependency analysis: First, it reconstructs the control flow graph of a contract and, based on that, transforms the contract to SSA form. The correctness of these steps is never discussed. Indeed we found Securify's algorithm for control flow reconstruction to be unsound: The algorithm fails when encountering jump destinations that depend on blockchain information. In such a case, the control flow should be considered to be non-reconstructable

[4] We illustrate the issue with a violation pattern for easier presentation and since the affected compliance pattern turned out not to be implemented in Securify.

since a jump to such a destination may result in a valid jump at runtime or simply fail due to a non-existing jump destination. Securify's algorithm, however, does not report an error on such a contract but returns a (modified) contract that does not contain jumps. Such an unsound preprocessing step again impacts the soundness of the whole analysis tool since it breaks the assumption that the contract semantics is preserved by preprocessing.

3.4.2 Security Properties

The Ethereum blockchain environment opens up several new attack vectors which are not present in standard (distributed) program execution environments. This is in particular due to the contracts' interaction with the blockchain, which is in general controlled by unknown parties and hence needs to be considered hostile. It is a (partly) still open research question of what characterizes a contract that is robust in such an environment. A well-studied property in this domain is robustness against reentrancy attacks, such as the DAO attack discussed in Chapter 1. We will focus on this property in the following to illustrate the challenges and pitfalls in proving a contract to be safe.

Formalizing Security Properties

While bug-finding tools typically make use of heuristics to detect vulnerable contracts, there have been two systematic studies that aim at giving a semantic characterization of what it means for a contract to be resistant against reentrancy attacks: The resulting security definitions are call integrity (defined in Section 2.4) and effective callback freedom [GAGG+17].

Call integrity follows non-interference-style integrity definitions from the security community. It states that two runs of a contract in which the codes of the environment accounts may differ should result in the same sequences of observable events (in this case, outgoing transactions). In simpler words, another contract should not be able to influence how a secure contract spends its money. Intuitively, this property is violated by the `DAO` contract presented in Fig. 1.1, since an attacker contract can make the contract send out more money than in an honest invocation.

In contrast, effective callback freedom is inspired by the concept of linearizability from concurrency theory: It should be possible to mimic every (recursive) execution of a contract by a sequence of non-recursive executions. The `DAO` contract violates this property since the attack is only possible when making use of recursion (or callbacks, respectively). After each callback-free execution, the `investments` mapping will be updated so that a subsequent execution will prevent further withdraws by the same party.

While we show in Section 2.4 how to over-approximate the hyperproperty call integrity by three simpler properties (the reachability property single-entrancy and two dependence properties), [GAGG+17] does not indicate a way of statically verifying effective callback freedom, but proves this property to be undecidable. This leaves sound and (efficiently) verifiable approximations an open research question.

```
1 library Lib {
2   struct Data { mapping (address => uint) map;}
3   function write(Data storage self, address a, uint v) { self.map[a] = v;}
4   function get(Data storage self, address a) returns (uint) {
5     return (self.map[a]);} }
6
7 contract DAO {
8   Lib.Data bal;
9   function invest() public payable {
10    Lib.write(bal, msg.sender, Lib.get(bal, msg.sender) + msg.value);}
11  function withdraw () public {
12    address a = msg.sender;
13    if (Lib.get(bal, a) > 0){
14      a.call.value(Lib.get(bal, a));
15      Lib.write(bal, a, 0);}} }
```

Figure 3.2: Simplified DAO contract using a library

Checking Security Properties

The state-of-the-art sound analyzers discussed so far do not build on prior efforts of semantically characterizing robustness against reentrancy attacks but come up instead with their own ad-hoc definitions.

Securify Securify expresses the security properties of smart contracts in terms of compliance and violation patterns over data flow and control flow dependency predicates. In [TDDC+18] it is stated that Securify supports the analysis of a property called 'no writes after call' (NW), which is different from (robustness against) reentrancy, but still aims at detecting bugs similar to the one in the `DAO`. The NW property is defined using an ad-hoc semantic formalism, and it states that for any contract execution trace, the contract storage shall not be subject to modifications after performing a CALL instruction. Intuitively, this property should exclude reentrancy attacks by preventing that the guards of problematic money transfers are updated only after performing the money transferring call. However, this criterion is not sufficient, e.g., since reentrancies can also be triggered by instructions other than CALL. For proving the NW property, the compliance pattern demands that a CALL instruction may not be followed by any SSTORE instruction. We found this pattern not to be sufficient for ensuring compliance with the NW property (nor robustness against reentrancy). We will illustrate this using a variation of the `DAO` contract in Figure 3.2. This contract implements the exact same functionality as the one in Figure 1.1. The only difference is that the access to the balance mapping is handled via the library contract `Lib`. Ethereum actively supports the use of library contracts by the DELEGATECALL instruction that executes another contract's code in the environment of the caller. When calling `Lib.write` in the `withdraw` function, such a delegated call to the (external) library contract is executed. Executing `write` in the context of contract `DAO` will then modify `DAO`'s storage (instead of the one of the `Lib` contract). In order to let the `write` and the `get` functionality access the right storage position (where `DAO` stores the `bal` mapping), these functions take as first argument the reference to the corresponding storage location. Same as the version in Figure 1.1, this contract is vulnerable to a reentrancy bug. Also, it violates the NW property: The storage of the contract

```
1 contract DAO{
2   mapping (address => uint) bal;
3   uint lock = 0;
4   function withdraw () public {
5     if(lock ==1){throw;}
6     lock=1;
7     address a = msg.sender;
8     a.call.value(bal[a]);
9     bal[a] = 0;
10    lock=0;}
11  function switchLock () {
12    lock = 1-lock;} }
```

```
1 contract DAO{
2   mapping (address => uint) bal;
3   uint lock = 0;
4   function withdraw () public {
5     if(lock ==1){throw;}
6     lock=1;
7     address a = msg.sender;
8     a.call.value(bal[a]);
9     bal[a] = 0;
10    lock=0;}
```

Figure 3.3: Simple versions of the DAO contract with reentrancy protection.

can be changed after executing the call (when writing the `bal`) mapping. Still, this contract matches the compliance pattern (which should, according to [TDDC+18] guarantee the contract to satisfy the NW property) since it does not contain any explicit SSTORE instruction. This example illustrates how without a proven connection between a property and its approximation, the soundness of an analyzer can be undermined. This issue does not only constitute a singular case but is a structural problem: There are counterexamples for the soundness of 13 out of the 17 patterns presented in [TDDC+18], as we detail out in Appendix B.1.

ZEUS In [KGDS18b], the property to rule out reentrancy attacks is only specified in prose as a function being vulnerable 'if it can be interrupted while in the midst of its execution and safely re-invoked even before its previous invocations complete execution.' This definition works on the level of functions, a concept which is only present on the *Solidity* level and leaves open the question of what it means for *a contract* to be robust against reentrancy attacks. The authors distinguish between 'same-function-reentrancy' and 'cross-function-reentrancy' attacks but do not consider cross-function reentrancy (where a function reenters another function of the same contract) in the analyzer. We found that without excluding cross-function reentrancy also single-function reentrancy cannot be prevented.

Consider the versions of the DAO contract depicted in Figure 3.3 that aim to prevent reentrancy using a locking mechanism. The global `lock` field tracks whether the `withdraw` function was already entered (indicated by value `1`). In that case, the execution of `withdraw` throws an exception. Otherwise, the `lock` is set and only released when concluding the execution of `withdraw`. While the two depicted contracts implement the exact same `withdraw` function, the first contract's function is vulnerable to a reentrancy attack, while the second one is safe. This is as the first contract implements a public `switchLock()` function that can be used by anyone to change the `lock` value. An attacker could hence mount the standard attack with the only difference that they would need to invoke the `switchLock()` function once before reentering to disable the reentrancy protection in line 5. Without exposing such functionality, the second contract is safe, since every reentering execution will be stopped in line 5. This example shows that ZEUS' approach of analyzing functions in isolation to exclude 'same-function-reentrancy' is not sound.

Another issue in the reentrancy checking of ZEUS is caused by the reentrancy property exceeding the scope of the analysis framework. For proving a function resistant against reentrancy attacks, ZEUS checks whether it is ever possible to reach a call when a function is recursively invoked by itself. However, the presented translation to LLVM bitcode only models non-recursive executions of a function. Consequently, the reentrancy property cannot be expressed as a policy (which could be translated to assertions in the program code) but requires rewriting the contract under analysis to contain duplicate functions that mimic reentering function invocations. This contract transformation is not part of any soundness considerations. As a result, not only the previously discussed unsoundness due to the lacking treatment of cross-function reentrancies is missed, but it is also disregarded that *Solidity*'s `call` construct is not the only way to reinvoke a function. Indeed there are several other mechanisms (e.g., direct function calls) that allow for the same functionality. Still, ZEUS classifies contracts that do not contain an explicit invocation of the `call` construct to be safe by default. More details are provided in Appendix B.1.5.

NeuCheck The NeuCheck tool formulates a syntactic pattern for detecting robustness against reentrancy attacks. The pattern checks for all occurrences of the `call` function whether they are followed by the assignment of a state variable. As discussed for Securify, the absence of explicit writes to the storage does not imply that the storage stays unchanged. Hence the example in Figure 3.2 would also serve as a counterexample for the soundness claim of NeuCheck. Also, as discussed for ZEUS, `call` is not the only way of invoking another contract, revealing another source of unsoundness in this definition. Furthermore, the authors neither specify the targeted security properties nor provide justifications for this syntactic analysis approach's soundness. For further details, we refer to Appendix B.1.3.

We summarize the soundness properties of the presented tools in Fig. 3.4. Note that we already include the tool *eThor* in this comparison which we will discuss in detail in Chapter 4. We report the soundness issues based on their occurrence in the analysis pipeline, distinguishing between the preprocessing, the core analysis, and the encoding of security properties.

3.5 Conclusion

We presented a systematization of the state of the art in Ethereum smart contract verification and outlined the open challenges in this field. In particular, we focused on the challenge of designing *automated and sound* static analyzers for EVM bytecode. To this end, we highlighted the peculiarities of the EVM bytecode language design and execution model that complicate the creation of such analyzers by showing how state-of-the-art analyzers fall short of providing reliable soundness guarantees. To show how to overcome these challenges in a principled fashion, in the next chapter, we will present *eThor*, the first automated and provable sound static analyzer for smart contracts written in EVM bytecode.

	Securify [TDDC+18]	**ZEUS** [KGDS18a]	**NeuCheck** [LWZ+19]	***eThor*** [SGSM20]
Targeted Language	EVM Bytecode	Solidity	Solidity	EVM Bytecode
Preprocessing	Unsound CFG Reconstruction *An example of a wrongly reconstructed CFG is given in Appendix B.1.1*	Unsound program transformation *Discussed in Section 3.4.1 and Appendix B.1.5*	No preprocessing	Sound CFG reconstruction *Formally proven in [Gri21]*
Core Analysis	Indication for unsoundness in dependence analysis *See example in Fig. 3.1*	Unsound modelling of reentering executions *Discussed in Section 3.4.1 and Appendix B.1.5*	No core analysis	Sound abstraction *Proven in Appendix C.2.4*
Security Properties	Non-sufficient dependence patterns *Counter examples for 13 out of 17 patterns are provided in Appendix B.1.2*	Non-sufficient (reachability) criteria *Discussed in Section 3.4.2 and Appendix B.1.5*	Non-sufficient syntactic patterns *Discussed in Section 3.4.2 and Appendix B.1.3*	Sound reachability criterion for single-entrancy *Proven in Appendix C.3.1*

Figure 3.4: Overview on the soundness guarantees and issues of the tools Securify, ZEUS, NeuCheck, and *eThor* broken down to the different phases of the analysis pipeline.

CHAPTER 4

eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts

Abstract

Ethereum has emerged as the most popular smart contract platform, with hundreds of thousands of contracts stored on the blockchain and covering diverse application scenarios, such as auctions, trading platforms, or elections. Given the financial nature of smart contracts, security vulnerabilities may lead to catastrophic consequences and, even worse, can hardly be fixed as data stored on the blockchain, including the smart contract code itself, are immutable. An automated security analysis of these contracts is thus of utmost interest but at the same time technically challenging. This is as, e.g., Ethereum's transaction-oriented programming mechanisms feature subtle semantics, and since the blockchain data at execution time, including the code of callers and callees, are not statically known.

In this work, we present *eThor*, the first sound and automated static analyzer for EVM bytecode, which is based on an abstraction of the EVM bytecode semantics based on Horn clauses. In particular, our static analysis supports reachability properties, which we show to be sufficient for capturing interesting security properties for smart contracts (e.g., single-entrancy) as well as contract-specific functional properties. Our analysis is proven sound against the EVM bytecode semantics presented in Chapter 2, and a large-scale experimental evaluation on real-world contracts demonstrates that *eThor* is practical and outperforms the state-of-the-art static analyzers: specifically, *eThor* is the only one to provide soundness guarantees, terminates on 94% of a representative set of real-world contracts, and achieves an F-measure (which combines sensitivity and specificity) of 89%.

This chapter presents the result of a collaboration with Ilya Grishchenko, Markus Scherer,

and Matteo Maffei and was published at the 27th The ACM Conference on Computer and Communications Security (CCS'20) under the title 'eThor*: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts' [SGSM20]. I am responsible for the formal development of the paper, including the analysis specification, the domain-specific abstractions, and proofs. Further, I conducted the case study and the investigation of the related work. In addition, I contributed to the design of the specification language* HoRSt*, as well as the design of the experimental setup. The implementation of* eThor*, as well as the compiler for the* HoRSt *language, was primarily done by Markus Scherer. The same applies to the experimental evaluation of* eThor*. The implementation of the control flow reconstruction pre-processing step was conducted by Ilya Grishchenko, who also contributed to the experimental evaluation. The accompanying appendix contains extended versions of the formalism and proofs.*

4.1 Introduction

Smart contracts introduced a radical paradigm shift in distributed computation, promising security in an adversarial setting thanks to the underlying consensus algorithm. Software developers can implement sophisticated distributed, transaction-based computations by leveraging the scripting language offered by the underlying blockchain technology. While many cryptocurrencies have an intentionally limited scripting language (e.g., Bitcoin [Nak08]), Ethereum was designed from the ground up with a quasi Turing-complete language[1]. Ethereum smart contracts have thus found a variety of appealing use cases, such as auctions [HSLC17], data management systems [Adh17], financial contracts [BKT17], elections [MFSH17], trading platforms [NGW17, MM17], permission management [AEVL16] and verifiable cloud computing [DWA+17], just to mention a few. Given their financial nature, bugs and vulnerabilities in smart contracts may lead to catastrophic consequences. For instance, the infamous DAO vulnerability [the16] recently led to a 60M$ financial loss, and similar vulnerabilities occur on a regular basis [par17a, par17b]. Furthermore, many smart contracts in the wild are intentionally fraudulent, as highlighted in a recent survey [ABC17]. Even worse, due to the unmodifiable nature of blockchains, bugs or vulnerabilities in deployed smart contracts cannot be fixed.

A rigorous security analysis of smart contracts is thus crucial for the trust of the society in blockchain technologies and their widespread deployment. Unfortunately, this task is quite challenging for various reasons. First, Ethereum smart contracts are developed in an ad-hoc language, called *Solidity*, which resembles *JavaScript* but features non-standard semantic behaviors and transaction-oriented mechanisms, which complicate smart contract development and verification. Second, smart contracts are uploaded to the blockchain in the form of Ethereum Virtual Machine (EVM) bytecode, a stack-based low-level code featuring very little static information, which makes it extremely difficult to analyze. Finally, most of the data available at runtime on the blockchain, including the contracts that the contract under analysis may interact with, may not be known statically, which requires ad-hoc abstraction techniques. As a result, while effective bug-finding tools for smart contracts have been recently presented, *there exists at present no*

[1] While the language itself is Turing complete, computations are bounded by a resource budget (called gas), consumed by each instruction thereby enforcing termination.

automated security analysis for EVM bytecode that provides formal security guarantees (i.e., absence of false negatives, as proven against a formal semantics of EVM bytecode) as detailed out in Chapter 3. Inspired by the prevalent issues in state of the art, we introduce a principled approach to the design and implementation of a sound yet performant, a static analysis tool for EVM bytecode.

4.1.1 Our Contributions

The contributions presented in this chapter can be summarized as follows:

- We design the first provably sound static analyzer for EVM bytecode, which builds on top of a Horn-clause-based reachability analysis. We show that reachability analysis suffices to verify interesting security properties for smart contracts as well as contract-specific functional properties via an encoding into Hoare-style reasoning. The design of such static analysis is technically challenging since it requires careful abstractions of various EVM components (e.g., the stack-based execution model, the gas bounding the smart contract execution, and the memory model) as well as a dedicated over-approximation of blockchain data, which are not statically known and yet affect contract execution (e.g., the code of other contracts which may act both as callers and callees);
- We prove our static analysis technique sound against the formal semantics of EVM bytecode presented in Chapter 2;
- In order to facilitate future refinements of our analysis, as well as the design of similar static analyses for other languages, we design and implement *HoRSt*, a framework for the specification and implementation of static analyses based on Horn clause resolution. Specifically, *HoRSt* takes as input a (mathematical) specification of the Horn clauses defining the static analysis and produces an `smt-lib` [smt20] encoding suitable for *z3* [HB12], which includes various optimizations such as Horn clause and constant folding;
- We use *HoRSt* to implement the static analyzer *eThor*. To gain confidence in the resulting implementation, we encode the relevant semantic tests (604 in total) of the official EVM suite as reachability properties, against which we successfully test the soundness and precision of *eThor*;
- We conduct a large-scale experimental evaluation on real-world contracts comparing *eThor* to the state-of-the-art analyzer ZEUS [KGDS18a] which claims to provide soundness guarantees. While ZEUS shows a striking specificity (i.e., completeness) of 99.8%, *eThor* clearly outperforms ZEUS in terms of recall (i.e., soundness) – 100% vs. 11.4% – which empirically refutes ZEUS' soundness claim. With a specificity of 80.4%, *eThor* shows overall performance of 89.1% (according to the F-measure) as compared to ZEUS' F-measure of 20.4%.

The remainder of this chapter is organized as follows. Section 4.2 introduces our static reachability analysis, specifies its soundness guarantee and discusses relevant smart contract properties in

scope of the analysis. Section 4.3 introduces the specification language *HoRSt*. Section 4.4 presents *eThor* and our experimental evaluation. Section 4.5 discusses how *eThor* overcomes the challenges discussed in Chapter 3. Section 4.6 concludes by discussing interesting future research directions. The source code of *eThor* and *HoRSt* with the data set used in the experimental evaluation are available online [ext20].

4.2 Static Analysis of EVM Bytecode

Starting from the small-step semantics presented in Chapter 2, we design a sound reachability analysis that supports (among others) the validation of the single-entrancy property. We follow the verification chain depicted in Figure 4.1: For showing the executions of a contract to satisfy some property Φ, we formulate a *Horn-clause-based abstraction* that abstracts the contract execution behavior and argue about an *abstracted property* over *abstract executions* instead. This reasoning is sound given that all concrete small-step executions are modeled by some abstract execution and given that the abstracted property over-approximates Φ.

A Horn-clause-based abstraction for a small-step semantics $\rightarrow$ is characterized by an abstraction function α that translates concrete configurations (here $\bullet$) into *abstract configurations* (here $\blacktriangle$). Abstract configurations are sets of predicate applications where predicates (formally characterized by their signature $\mathcal{S}$) range over the values from abstract domains. These abstract arguments are equipped with an order $\leq$ that can be canonically lifted to predicates and further to abstract configurations, hence establishing a notion of precision on the latter. Intuitively, α translates a concrete configuration into its most precise abstraction. The *abstract semantics* is specified by a set of Constrained Horn clauses Λ over the predicates from $\mathcal{S}$ and describes how abstract configurations evolve during abstract execution. A Constrained Horn clause is a logical implication that can be interpreted as an inference rule for a predicate. Consequently, an abstract execution consists of logical derivations from an abstract configuration using Λ. A Horn-clause-based abstraction constitutes a sound approximation of small-step semantics $\rightarrow$ if every concrete (multi-step) execution $\bullet \rightarrow^* \bullet'$ can be simulated by an abstract execution: More precisely, from the abstract configuration $\alpha(\bullet)$ one can logically derive using Λ an abstract configuration $\blacktriangle$ that constitutes an over-approximation of $\bullet'$ (so is at least as abstract as $\alpha(\bullet')$). A formal presentation of the soundness statement is given in Section 4.2.4 while a characterization in abstract interpretation terminology is deferred to Appendix C.2.2. A sound abstraction allows for the provable analysis of *reachability properties*: Such properties can be expressed as sets of problematic configurations (here $\bullet$). Correspondingly, a sound abstraction for such a property is a set of bad abstract configurations (here $\blacktriangle$) that contains all possible over-approximations of the bad concrete states. The soundness of the abstract semantics then guarantees that if no bad abstract configuration from this set can be entered, also no bad configuration can be reached in the concrete execution.

4.2.1 Main Abstractions

Our analysis abstracts from several details of the original small-step semantics. In the following, we overview the main abstractions:

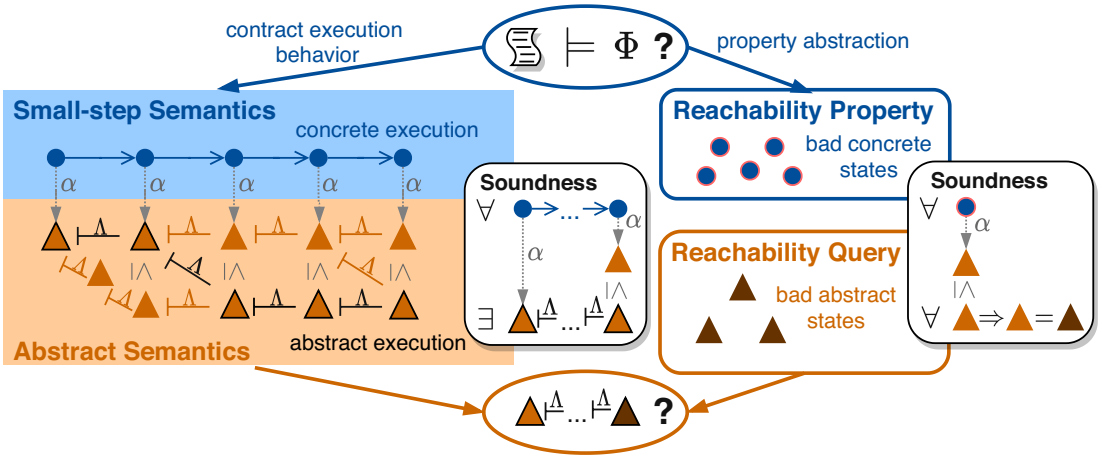

Figure 4.1: Formal verification chain of *eThor*. $\Delta \vdash^{\Lambda} \Delta'$ denotes that the abstract configuration Δ' can be logically derived from Δ' (within one step) using the Horn clauses in Λ.

Blockchain environment. The analysis describes the invocation of a contract (in the following denoted as c^*) in an arbitrary blockchain environment, hence is not modeling the execution environment as well as large fractions of the global state. Indeed, most of this information is not statically known as the state of the blockchain at the time of contract execution cannot be reliably predicted. As a consequence, the analysis has to deal with a high number of unknown environment inputs in the abstract semantics. Most prominently, the behavior of other contracts needs to be appropriately over-approximated, which turns out to be particularly challenging since such contracts can interact with c^* in multitudinous ways.

Gas modeling. The contract gas consumption is not modeled. The gas resource bounding the contract execution is set by the transaction initiator and hence not necessarily known at analysis time. Thus, our analysis assumes contract executions to halt exceptionally at any point due to an out-of-gas exception. This does not affect the precision of the analysis for security properties that consider arbitrary contract invocations (and hence arbitrary gas limits).

Memory model. In the EVM the local memory is byte-indexed, while the machine stack holds words (encompassing 32 bytes). Consequently, loading a machine word from memory requires assembling the byte values from 32 consecutive memory cells. However, as already described in [PZS+18], in practice, reasoning about this conversion between words and bytes is hard. Therefore, we model memory in our abstraction as a word array: this enables very cheap accesses in case that memory is accessed at the start of a new memory word and otherwise just requires the combination of two memory words.

Call Stack. The call stack is captured by a two-level abstraction distinguishing only between the original execution of c^* (call level 0) and reentrancies of c^* ultimately originating from the original execution (call level 1). This abstraction reflects that given the unknown blockchain environment, the state of the call stack when reentering is obscure: it is unknown who initiated the reentering call and which other internal transactions have been executed before.

$$
\begin{array}{rcl}
\mathcal{S}_{c^*} & \ni p & := \\
 & \mid & \mathsf{MState}_{\mathsf{pc}} : (\mathbb{N} \times (\mathbb{N} \to \hat{D})) \times (\mathbb{N} \to \hat{D}) \times (\mathbb{N} \to \hat{D}) \times \mathbb{B} \to \mathbb{B} \\
 & \mid & \mathsf{Exc} : \mathbb{B} \to \mathbb{B} \\
 & \mid & \mathsf{Halt} : (\mathbb{N} \to \hat{D}) \times \mathbb{B} \to \mathbb{B} \\
 & \mathsf{pc} \in & \{0, \ldots, |c^*.\mathit{code}| - 1\} \\
\hat{D} & := & \mathbb{N} \cup \{\top\}
\end{array}
$$

Figure 4.2: Definition of the predicate signature $\mathcal{S}_{c^*}$ and the abstract domain $\hat{D}$.

4.2.2 Analysis Definition

In the following we formally specify our analysis by defining the underlying Horn-clause-based abstraction. An abstract configuration is a set of predicate applications representing one or several concrete configurations. Since we are interested in analyzing executions of the contract c^*, we consider EVM configurations representing such executions which are call stacks having an execution state of contract c^* as a bottom element. We abstract such a call stack by the set of all its elements that describe executions of c^*, reflecting the stack structure only by indicating whether a relevant execution state represents the original execution of c^* (call level 0) or a reentering execution that hence appears higher on the call stack (call level 1). The individual execution states are abstracted as predicate applications using the predicates listed in Figure 4.2: A predicate application of the form $\mathsf{MState}_{\mathsf{pc}}((\mathit{size}, s), m, \mathit{stor}, c\ell)$ describes a regular execution of c^* at program counter pc that has a local stack of size *size* with elements as described by the mapping s (from stack positions to elements) and a local memory m, and the global storage of contract c^* at this point being *stor*. Accordingly, the predicate application $\mathsf{Exc}(c\ell)$ denotes that an execution of c^* halted exceptionally at call level $c\ell$ and $\mathsf{Halt}(\mathit{stor}, c\ell)$ represents an execution that regularly halted at call level $c\ell$ with the global storage of c^* being *stor*. Since during the abstract execution, precise modeling of all the described state components is not always possible, the argument domains of the predicates encompass the abstract domain $\hat{D}$ that enriches $\mathbb{N}$ with the join element $\top$ over-approximating any natural number. Formally, the described abstractions of EVM configurations are captured by the abstraction function α in Figure 4.3 that maps call stacks into the corresponding sets of predicates, yielding an abstract configuration.

Note that α is parametrized by c^* and that only the call stack elements modeling executions of c^* are translated.

The transitions between abstract configurations (as yielded by α) are described by an abstract semantics in the form of Constrained Horn clauses. The abstract semantics is also specific to the contract c^*: Depending on the EVM instructions that appear in c^*, it contains Horn clauses that over-approximate the execution steps enabled by the corresponding instructions. We hence formulate the abstract semantics as a function δ that maps a contract c^* to the union of Horn clauses that model the individual instructions in the contract:

$$\alpha_{c^*}(S) := \begin{cases} \emptyset & S = \epsilon \\ \alpha_s(s, c^*.addr, c\ell) \cup \alpha_{c^*}(S') & S = s_{c^*} :: S' \ \wedge \ c\ell = (S' = \epsilon)\,?\,0 \,:\, 1 \\ \alpha_{c^*}(S') & S = s_c :: S' \ \wedge \ c \neq c^* \end{cases}$$

$$\alpha_s(s, a, c\ell) := \begin{cases} \{\mathsf{MState}_{pc}((|s|, \\ \quad \mathsf{stackToArray}(s)), \\ \quad \mathsf{toWordMem}(m), \\ \quad \sigma(a).stor, c\ell)\} & s = ((gas, pc, m, i, s), \iota, \sigma, \eta) \\ \{\mathsf{Exc}(c\ell)\} & s = EXC \\ \{\mathsf{Halt}(\sigma(a).stor, c\ell)\} & s = HALT(\sigma, gas, data, \eta) \\ \emptyset & otherwise \end{cases}$$

$$\mathsf{stackToArray}(s) := \begin{cases} \lambda x.\, 0 & s = \epsilon \\ (\mathsf{stackToArray}(s'))_x^{|s'|} & s = x :: s' \end{cases}$$

$$\mathsf{toWordMem}(m) := \lambda x.\, m[x \cdot 32]||_1 m[x \cdot 32 + 1] \cdots ||_1 m[x \cdot 32 + 31]$$

Figure 4.3: Configuration abstraction function. Here $v||_n w$ denotes the value obtained by concatenating v's and w's byte representation, assuming that w is represented by n bytes.

$$\delta(c^*) := \bigcup_{0 \leq i < |c^*.code|} (\!|c^*.code[i]|\!)_i$$

The core of the abstract semantics is defined by the *instruction abstraction function* $(\!|\cdot|\!)_i$ that maps a contract instruction at position i to a set of Horn clauses over-approximating the semantics of the corresponding instruction. We will discuss the translation of the ADD, the MLOAD, and CALL instruction depicted in Figure 4.4 to illustrate the main features of the abstract semantics.

Addition. The abstract semantics of the addition instruction (ADD) encompasses two Horn clauses describing the successful execution and the failure case. A prerequisite for a successful addition is the existence of a sufficient amount of arguments on the machine stack. In this case, the top stack values are extracted, and the stack at the next program counter (modeled by the predicate $\mathsf{MState}_{\mathsf{pc}+1}$) is updated with their sum. As the stack elements, however, range over the abstract value domain $\hat{D}$, the addition operation on $\mathbb{N}$ needs to be lifted to $\hat{D}$: Following the general intuition of $\top$ representing all potential values in $\mathbb{N}$, the occurrence of $\top$ as one of the operands immediately declassifies the result to $\top$. Similar liftings are performed for all unary, binary, and comparison operators in the instruction set. A precise definition is given in Appendix C.2.3.

In accordance with the choice of not modeling gas consumption, the Horn clause modeling the failure case – which is common to the abstract semantics of all instructions – does not have any preconditions, but the instruction reachability. This rule subsumes all other possible failure cases (such as stack over- and underflows).

$$
\begin{aligned}
&(\!|\mathsf{ADD}|\!)_{\mathsf{pc}} := \\
&\quad \{\, \mathsf{MState}_{\mathsf{pc}}((\mathit{size}, s), m, \mathit{stor}, c\ell) \ \wedge\ \mathit{size} > 1 \\
&\quad\ \wedge\ \hat{x} = s[\mathit{size}-1] \ \wedge\ \hat{y} = s[\mathit{size}-2] \\
&\quad\ \Longrightarrow\ \mathsf{MState}_{\mathsf{pc}+1}((\mathit{size}-1, s[\mathit{size}-2 \to \hat{x} \mathbin{\widehat{+}} \hat{y}]), m, \mathit{stor}, c\ell), && \text{(A1)} \\
&\quad \mathsf{MState}_{\mathsf{pc}}((\mathit{size}, s), m, \mathit{stor}, c\ell) \ \Longrightarrow\ \mathsf{Exc}(c\ell)\,\} && \text{(A2)} \\
&(\!|\mathsf{MLOAD}|\!)_{\mathsf{pc}} := \\
&\quad \{\, \mathsf{MState}_{\mathsf{pc}}((\mathit{size}, s), m, \mathit{stor}, c\ell) \ \wedge\ \mathit{size} > 1 \\
&\quad\ \wedge\ \hat{o} = s[\mathit{size}-1] \ \wedge \hat{v} = (\hat{o} \in \mathbb{N})\,?\,\mathit{getWord}(m, \hat{o}) \ :\ \top \\
&\quad\ \Longrightarrow\ \mathsf{MState}_{\mathsf{pc}+1}((\mathit{size}, s[\mathit{size}-1 \to \hat{v}]), m, \mathit{stor}, c\ell), \ldots\,\} && \text{(M1)} \\
&(\!|\mathsf{CALL}|\!)_{\mathsf{pc}} := \\
&\quad \{\, \mathsf{MState}_{\mathsf{pc}}((\mathit{size}, s), m, \mathit{stor}, c\ell) \ \wedge\ \mathit{size} > 6 \\
&\quad\ \Longrightarrow\ \mathsf{MState}_{\mathsf{pc}+1}((\mathit{size}-6, s[\mathit{size}-7 \to \top]), \lambda x.\,\top, \lambda x.\,\top, c\ell), && \text{(C1)} \\
&\quad \mathsf{MState}_{\mathsf{pc}}((\mathit{size}, s), m, \mathit{stor}, c\ell) \ \wedge\ \mathit{size} > 6 \\
&\quad\ \Longrightarrow\ \mathsf{MState}_{0}((0, \lambda x.\,0), \lambda x.\,0, \mathit{stor}, 1), && \text{(C2)} \\
&\quad \mathsf{MState}_{\mathsf{pc}}((\mathit{size}, s), m, \mathit{stor}, c\ell) \ \wedge\ \mathit{size} > 6 \ \wedge\ \mathsf{Halt}(\mathit{stor}_h, 1) \\
&\quad\ \Longrightarrow\ \mathsf{MState}_{0}((0, \lambda x.\,0), \lambda x.\,0, \mathit{stor}_h, 1), \ldots\} && \text{(C3)}
\end{aligned}
$$

Figure 4.4: Partial definition of $(\!|\cdot|\!)_{\mathsf{pc}}$: selection of abstract semantics rules. For CALL and MLOAD the exception rule is omitted.

Memory Access. Memory access on the level of EVM bytecode is enabled by the MLOAD instruction, which takes the memory offset to be accessed as the argument from the stack and pushes the word from memory starting at this index. In our abstraction defined by the abstract semantic rule depicted in Figure 4.4 either immediately $\top$ is pushed to the stack (in case that the offset $\hat{o}$ is not a concrete value and hence the value to be loaded cannot be determined) or the word from the concrete memory offset is extracted. The extraction needs to account for the word-indexed memory abstraction that we chose and is formally defined by the function $\mathit{getWord}(\cdot,\cdot)$ depicted in Figure 4.5. In case that the offset is a word address (divisible by 32), the corresponding value can be accessed from the word memory m by converting the byte address to the word address ($\frac{p}{32}$). Otherwise, the word at the next lower word address ($\lfloor \frac{p}{32} \rfloor$) and the word at the next higher byte address ($\lceil \frac{p}{32} \rceil$) are accessed to combine their relevant parts to a full word.

Contract Calls. The abstraction for CALL is the most interesting. This instruction takes seven arguments from the stack that specify parameters to the call, such as the target of the call or the value to be transferred, as well as the memory addresses specifying the location of the input and the return data. When returning from a successful contract call, the value 1 is written to the stack, and the return value is written to the specified memory fragment. The persistent storage after a successful call contains all changes that were performed during the execution of the called contract. In the case of an exception, the storage is rolled back to the point of calling and the

$$\mathit{getWord}(m,p) := \begin{cases} m[\frac{p}{32}] & p \bmod 32 = 0 \\ (m[\lfloor \frac{p}{32} \rfloor]_{[p \bmod 32,31]})||_{p \bmod 32}(m[\lceil \frac{p}{32} \rceil]_{[0,(p \bmod 32)-1]}) & \text{otherwise} \end{cases}$$

Figure 4.5: Function extracting the word at byte offset p from word-indexed memory m. Here $v_{[l,r]}$ denotes the value represented by v's lth byte till rth byte in big endian byte representation. $v||_n w$ is defined as in Figure 4.3. We assume both operations to be lifted to $\hat{D}$.

value 0 is written to the stack to indicate failure.

Since a contract call initiates the execution of another (unknown) contract, all its effects on the executions of c^* need to be modeled. More precisely, these effects are two-fold: the resuming execution of c^* on the current call level needs to be approximated, as well as the reentering executions of c^* (on a higher call level). For obtaining an analysis that is precise enough to detect real-world contracts with reentrancy protection as secure, it is crucial to model c^*'s persistent storage as accurately as possible in reentering executions. This makes it necessary to carefully study how the storage at the point of reentering relates to the one in the previous executions of c^*, taking into account that (in the absence of DELEGATECALL and CALLCODE instructions in c^*) only c^* can manipulate its own storage. Figure 4.6 overviews the storage propagation in the case of a contract call: To this end it shows the sequence diagram of a concrete execution of c^* that calls a contract c' which again triggers several reentrancies of c^*. In this course three ways of storage propagation between executions of c^* are exhibited: 1) The storage is *forward propagated* from a calling execution to a reentering execution of c^* (A, C) 2) The storage is *cross propagated* from a finished reentering execution to another reentering execution of c^* (B) 3) The storage is *back propagated* from a finished reentering execution to a calling execution of c^* (D, E) These three kinds of propagation are reflected in the three abstract rules for the call instruction given in Figure 4.4 and correspondingly visualized in Figure 4.6.

Rule (C1) describes how the execution of c^* (original and reentering alike) resumes after returning from the call, and hence approximates storage backpropagation: For the sake of simplicity, storage gets over-approximated in this case by $\lambda x.\, \top$. The same applies to the local memory, and stack top value as those are affected by the result of the computation of the unknown contract. Rule (C2) captures the initiation of a reentering execution (at call level 1) with storage forward propagation: As contract execution always starts at program counter 0 with empty stack and all-zero local memory, only abstractions (instances of the MState_0 predicate) of this shape are implied. The forward propagation of storage is modeled by initializing the MState_0 predicate with the storage *stor* at call time. Rule (C3) models storage cross propagation: Similar to rule (C2), an abstract reentering execution in a fresh machine state is triggered. However, the storage is not propagated from the point of calling, but from a finished reentering execution whose results are abstracted by the halting predicate Halt at call level 1. This rule is independent of the callee in that it is only conditioned on the reachability of some CALL instruction, but it does not depend on the callee's state. Its cyclic structure requires extrapolating an invariant on the potential storage modifications that are computable by c^*: Intuitively, when reentering c^*, it needs to be considered

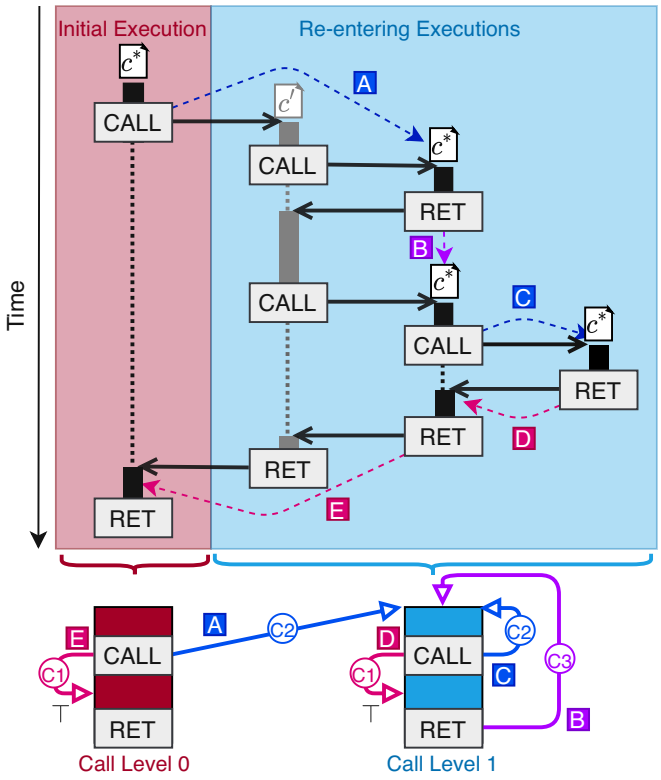

Figure 4.6: Illustration of the different call abstractions.

that priorly the storage was modified by applying an arbitrary sequence of c^*'s public functions. The significance of this abstraction is motivated by the example in Fig. 3.3 where for the `DAO` contract that supports the `switchLock()` function, the attack is only enabled by calling this function first to release the lock before reentering.

4.2.3 Scope of the Analysis

Before presenting the soundness result, we discuss the scope of the analysis. The analysis targets contracts in a stand-alone setting, which means that the behavior of all contracts that c^* might interact with is over-approximated. This abstraction is not merely a design choice but rather a necessity as the state of the blockchain (including the code of the contracts residing there) at execution time cannot be statically determined. Still, we could easily accommodate the precise analysis of a set of known contracts, e.g., library contracts that are already present on the blockchain. We omitted this straightforward extension in this work for the sake of clarity and succinctness in the analysis definition and the soundness claim.

Following this line of argumentation, we assume c^* not to contain DELEGATECALL and CALLCODE instructions: these instructions enable the execution of another contract code in the context of c^*, allowing for the modification of the persistent storage of c^* and even of money

transfers on behalf of c^*. Using DELEGATECALL or CALLCODE to call an unknown contract can, therefore, potentially result in the reachability of any execution states of contract c^*. Consequently, every property relying on the non-reachability of certain problematic contract states would be considered violated. In a setting of multiple known contracts, the restriction on DELEGATECALL and CALLCODE instructions could be relaxed to allow for such calls that are guaranteed to target known contracts.

We now briefly illustrate the key design choices behind our abstraction, which we carefully crafted to find the sweet spot between accuracy and practicality. The analysis is value-sensitive in that concrete stack, memory, and storage values are tracked until they get abstracted due to the influence of unknown components. For local computations, the analysis is partly flow-sensitive (considering the order of instructions but merging abstract configurations at the same program counters) and path-sensitive (being sensitive to branch conditions). On the level of contract calls, a partial context-sensitivity is given in that the storage at the time of calling influences the analysis of the subsequent call, but no other inputs to the call are tracked. In particular (due to the lack of knowledge on interactions with other contracts), all reentering calls are merged into a single abstraction, accumulating all possible storage states at the point of reentering. For this reason, the analysis of calls on level 1 is less precise than the one of the original execution at call level 0, where only the restrictions of flow sensitivity apply.

4.2.4 Soundness Result

We prove, for each contract c^*, that the defined Horn-clause-based abstraction soundly over-approximates the small-step semantics presented in Chapter 2. Formally, this property is stated as follows:

Theorem 3 (Soundness)**.** *Let c^* be a contract not containing DELEGATECALL or CALLCODE. Let Γ be a transaction environment and let S and S' be annotated call stacks such that $|S'| > 0$. Then for all execution states s that are strongly consistent with c^* it holds that*

$$\begin{aligned}\Gamma \vDash s_{c^*} :: S \rightarrow^* S' +\!\!+ S \implies \forall \Delta_I.\, \alpha_{c^*}([s_{c^*}]) \leq \Delta_I \\ \implies \exists \Delta.\, \Delta_I, \delta(c^*) \vdash \Delta \ \wedge\ \alpha_{c^*}(S') \leq \Delta\end{aligned}$$

Note that the notion of strong consistency here simply ensures that the contract annotation properly reflects the contract in the global state and in the execution environment. The formal details are given in Appendix C.2.4.

The theorem states that every execution of contract c^* (modeled by a multi-step execution starting in state s_{c^*} on an arbitrary call stack S and ending in call stack $S' +\!\!+ S$, indicating that the original execution of c^* yielded the state as modeled by the call stack S'), can be mimicked by an abstract execution. This means that from every abstract configuration Δ_I that abstracts s_{c^*} (so that it is more abstract than $\alpha([s_{c^*}])$) one can logically derive using the Horn clauses in $\delta(c^*)$ some abstract configuration Δ abstracting S'. As a consequence of this theorem, we can soundly reason about arbitrary executions of a contract c^*: if we can show that from an abstract configuration Δ_I, that abstracts a set of initial execution states of c^*, it is impossible to derive

using $\delta(c^*)$ some other abstract configuration Δ, that abstracts a set of problematic execution states of c^*, then this ensures that all these problematic states are not reachable with a small-step execution from any of the initial states.

For the proof of Theorem 3 we refer the reader to Appendix C.2.4.

4.2.5 Reachability Properties for Contract Safety

As characterized by the soundness result, our abstraction allows for the sound analysis of reachability properties. We will illustrate in the following how such a reachability analysis is sufficient to express relevant smart contract security properties.

Single-entrancy. Some generic security properties of Ethereum smart contracts can be over-approximated by reachability properties and thus checked automatically by our static analysis. Consider, the single-entrancy property from Section 2.4 which can be proven to be approximated by the following reachability property:

Definition 10 (Call unreachability)**.** *A contract c is call unreachable if for all regular execution states $(\mu, \iota, \sigma, \eta)$ that are strongly consistent with c and satisfy $\mu = (g, 0, \lambda x.\, 0, 0, \epsilon)$ for some $g \in \mathbb{N}$, it holds that for all transaction environments Γ and all call stacks S*

$$\begin{aligned}\neg\exists s, S.\, \Gamma \vDash (\mu, \iota, \sigma, \eta)_c :: S \;\rightarrow^*\; s_c :: S' +\!\!+ S \\ \wedge\; |S'| > 0 \;\wedge\; \mathit{code}\,(c)\,[s.\mu.\mathsf{pc}] \in \mathit{Inst}_{\mathit{call}}\end{aligned}$$

Where $\mathit{Inst}_{\mathit{call}} = \{\mathsf{CALL}, \mathsf{CALLCODE}, \mathsf{DELEGATECALL}, \mathsf{CREATE}\}$

Intuitively, call unreachability is a valid over-approximation of single-entrancy as an internal transaction can only be initiated by the execution of a call instruction. Consequently, for excluding that an internal transaction was initiated after reentering, it is sufficient to ensure that no call instruction is reachable at this point. In addition, as all contracts start their executions in a fresh machine state (program counter and active words set to 0, empty stack, memory initialized to 0) when being initially called, it is sufficient to check all executions of contract c that started in such a state. The formal proof for call unreachability implying single-entrancy is deferred to Appendix C.3.1.

Static Assertion Checking. The *Solidity* language supports the insertion of assertions into source code. Assertions shall function as pure sanity checks for developers and are enforced at runtime by the compiler creating the corresponding checks on the bytecode level and throwing an exception (using the INVALID opcode) in case of an assertion violation [Eth18]. However, adding these additional checks creates a two-fold cost overhead: At create time, a longer bytecode needs to be deployed (the longer the bytecode, the higher the gas cost for creation), and at call time, the additional checks need to be executed, which results in additional gas consumption. With our static analysis technique, assertions can be statically checked by querying the reachability of the INVALID instruction. If no such instruction is reachable, by the soundness of the analysis, the code is proven to give the same guarantees as with the assertion checks (up to gas), and those

checks can safely be removed from the code resulting in shorter and cheaper contracts.[2] Formally, we can characterize this property as the following reachability property:

Definition 11 (Static assertion checking)**.** *Let c be a contract and $(\mu, \iota, \sigma, \eta)$ regular execution states such that $(\mu, \iota, \sigma, \eta)$ is strongly consistent with c and $\mu = (g, 0, \lambda x.\, 0, 0, \epsilon)$ for some $g \in \mathbb{N}$. Let Γ be an arbitrary transaction environment and S be an arbitrary call stack. Then a the static assertion check for c is defined as follows:*

$$\neg\exists s, S.\, \Gamma \vDash (\mu, \iota, \sigma, \eta)_c :: S \;\rightarrow^* s_c :: S' +\!\!+ S \;\wedge\; \mathit{code}\,(c)\,[s.\mu.\mathsf{pc}] = \mathit{INVALID}$$

Intuitively this property says that during the execution of contract c, it should never be possible to execute an INVALID instruction.

Semi-automated Verification of Contract-specific Properties. As demonstrated by Hildebrandt et al. [HSR+18], reachability analysis can be effectively used for Hoare-Logic-style reasoning. This holds in particular for the analysis tool presented in this work: Let us consider a Hoare triple $\{P\}C\{Q\}$ where P is the precondition (operating on the execution state), C is the contract code, and Q is the postcondition that should be satisfied after executing code C in an execution state satisfying P. Then we can intuitively check this claim by checking that a state satisfying $\neg Q$ can never be reached when starting execution in a state satisfying P. More formally, we can define Hoare triples as reachability properties as follows:

Definition 12 (Hoare triples)**.** *Let c^* be a contract and let C be a code fragment of c^*. Let $P \in \mathcal{S} \rightarrow \mathbb{B}$ be a predicate on execution states (strongly consistent with c^*) that models execution right at the start of C and similarly let $Q \in \mathcal{S} \rightarrow \mathbb{B}$ be a predicate on execution states (strongly consistent with c^*) that models execution right at the point after executing C. Then Hoare triples $\{P\}C\{Q\}$ can be characterized as follows:*

$$\{P\}C\{Q\} := \forall s.\, P(s) \implies \neg\exists s'.\, \Gamma \vDash s_{c^*} :: S \;\rightarrow^*\; s'_{c^*} :: S \;\wedge\; \neg Q(s')$$

Hoare-Logic style reasoning can be used for the semi-automated verification of smart contracts, given that their behavior is specified in terms of pre- and postconditions. For now, it still requires a non-negligible amount of expertise to insert the corresponding abstract conditions on the bytecode-level, but by a proper integration into the *Solidity* compiler, the generation of the initialization and reachability queries could be fully automated (cf. Appendix C.3.2). We want to stress that in contrast to existing approaches, our analysis technique has the potential to provide fully automated pre- and postcondition checking even in the presence of loops as it leverages the fixed point engines of state-of-the-art SMT solvers [HBDM11].

4.3 *HoRSt*: A Static Analysis Language

To facilitate the principled and robust development of static analyzers based on Horn clause resolution, we designed *HoRSt* – a framework consisting of a high-level specification language

[2] The *Solidity* Docs [Eth18] discuss exactly this future use of static analysis tools for assertion checking.

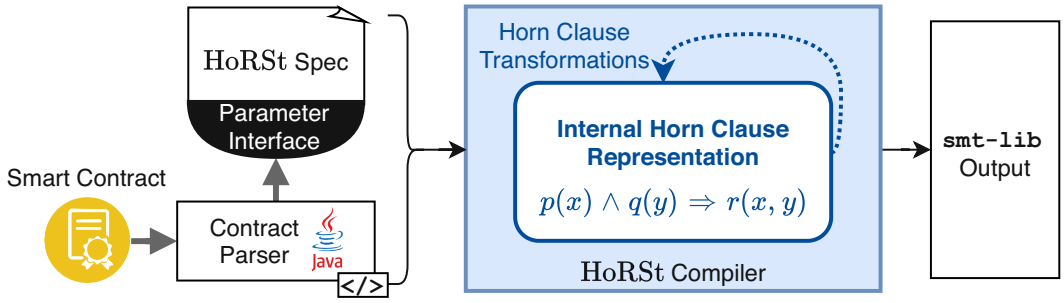

Figure 4.7: Utilization of *HoRSt* for static analysis

for defining Horn-clause-based abstractions and a compiler generating optimized `smt-lib` encodings for SMT solvers. The objective of *HoRSt* is to assist analysis designers in developing fast and robust static analyzers from clean and readable logical specifications.

Many existing practical analyzers are built on top of modern SMT solvers, such as *z3* . These solvers are highly optimized for performance. However, they show big performance deviations on different problem instances, which are (due to the heavy use of heuristics) difficult to predict for the users. Handcrafting logical specifications for such solvers in their low-level input format `smt-lib` hence not only cumbersome, error-prone, and requires technical expertise, but is also very inflexible, since the performance effects of different encodings may vary with the concrete problem instance. For tackling this issue, *HoRSt* decouples the high-level analysis design from the compilation to the input format: A high-level specification format allows for clear, human-readable analysis definitions while the translation process is handled by a stable and streamlined backend. On top, *HoRSt* allows for easily applying and experimenting with different Horn-clause-level optimizations that we can show to enhance the performance of *z3* substantially in our problem domain. We will shortly illustrate the utilization of *HoRSt* in the design process of our static analyzer and discuss the most interesting optimizations performed by the *HoRSt* compiler. For an introduction to the *HoRSt* language, we refer the reader to Appendix C.1.1.

Designing Static Analyses using *HoRSt* The *HoRSt* language allows for writing math-like specifications of Horn clauses such as those given in Figure 4.4. For parametrizing those clauses (e.g., by the program counters of a specific contract), an interface with a *Java*™ back end can be specified to handle the domain-specific infrastructure, such as contract parsing. We overview the different steps of the analysis design process in Fig. 4.7.

The core of the analysis is the *HoRSt* specification. Using high-level programming constructs such as algebraic data types, pattern matching, and bounded iteration, a *HoRSt* specification describes Constrained Horn clauses over user-defined predicates. Horn clauses can be parametrized by (families) of sets that are specified in the parameter interface (e.g., the sets of all program counters containing a certain bytecode instruction in a specific contract). Given such a specification, the analysis designer needs to provide infrastructure code written in *Java*™. In particular, this code needs to exhibit an implementation of those sets (or functions) specified in the parameter interface. In the case of our analysis, the environment code contains the infrastructure for contract parsing,

and the parameter interface allows for accessing the assembled contract information (code length, positions of opcodes, etc.) in the analysis specification. The *HoRSt* compiler itself is utilized to generate (optimized) `smt-lib` output given a *HoRSt* specification and the parameter interface implementation: It unfolds the high-level specification into separate Horn clauses over basic data types, applying the interface implementation. To this end, it also resolves all high-level constructs, ensuring that the resulting Horn clauses fall into the fragment that can be handled by *z3* . On top, the *HoRSt* compiler (optionally) performs different optimizations and transformations on the resulting Horn clauses before translating them into the standardized SMT output format `smt-lib`. The most important of these transformations are discussed in the following.

Low-level Optimizations One of the most effective optimizations performed by *HoRSt* is the predicate elimination by *unfolding* Horn clauses. This satisfiability preserving transformation has been long-studied in the literature [BD77, Tam84] and showed beneficial for solving Horn clauses in certain settings [HBC+12, BGMR15]. In practice, however, the exhaustive application of this transformation can lead to an exponential blow-up in the number of Horn clauses and hence does not necessarily yield the best results. For this reason *HoRSt* implements different strategies for the (partial) application of this transformation, which we call *linear folding* and *exhaustive folding*.

Finally, *HoRSt* supports constant folding for minimizing the `smt-lib` output and value encoding to map custom data types into primitive type encodings that are efficiently solvable by *z3* . We refer to Appendix C.1 for further details on *HoRSt* internals and functionalities.

4.4 Implementation & Evaluation

We use *HoRSt* to generate the analyzer *eThor* which implements the static analysis defined in Section 4.2. In the following, we overview the design of *eThor* and illustrate how *eThor* can enhance smart contract security in practice. To this end, we conduct a case study on a widely used library contract, showing *eThor*'s capability of verifying functional correctness properties and static assertion checks. Further, we validate *eThor*'s soundness and precision on the official EVM test suite and run a large-scale evaluation for the single-entrancy property on a set of real-world contracts from the Ethereum blockchain, comparing *eThor* with the state-of-the-art analyzer ZEUS [KGDS18a].

4.4.1 Static Analysis Tool

The mechanics of *eThor* are outlined in Fig. 4.8: *eThor* takes as input the smart contract to be analyzed in bytecode format and a *HoRSt* specification parametrized by the contract. To enhance the tool's performance and precision, *eThor* performs a multi-staged analysis: First, it approximates the contract jump destinations. This step decouples the control flow reconstruction (which can be performed more efficiently with a less precise abstract semantics as typically no computations on jump destinations are performed, but just their flow during the stack needs to be modeled) from the more evolved abstract semantics required for precisely analyzing the properties discussed in Section 4.2.5. As both used semantics are sound, the soundness of the

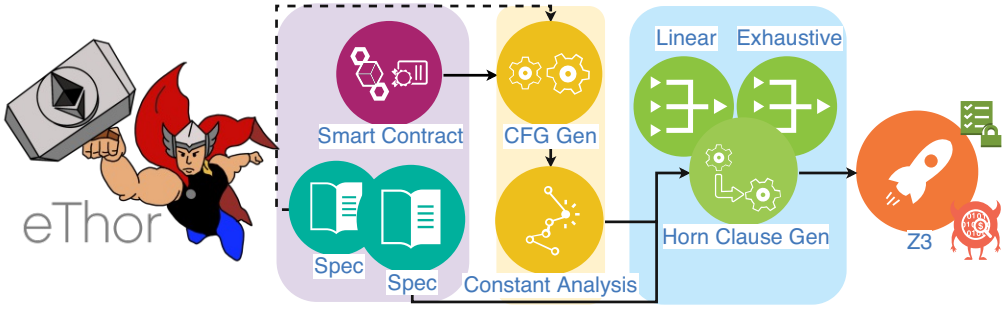

Figure 4.8: Analysis outline.

overall analysis is guaranteed. In a second preprocessing step, *eThor* performs a simple partial execution of atomic program blocks in order to statically determine fixed stack values. This can be beneficial in order to, e.g., precompute hash values and results of exponentiation which would otherwise need to be over-approximated in the analysis due to the lacking support for such operations by *z3* . The results from the preanalysis steps are incorporated into the analysis by a predefined interface in the *HoRSt*-specification. The *HoRSt* compiler then – given the interface implementation and the specification – creates an internal Horn clause representation which, after optionally performing different optimizations, is translated to an `smt-lib` file on which the SMT solver *z3* is invoked. The reconstructed control flow is obtained by a *Soufflé* [JSS16] program, which was created by manually translating a *HoRSt* specification. *Soufflé* is a high performance datalog engine, which we plan to support as a compilation target for (a subset of) *HoRSt* in the near future.z Since the problem of control flow reconstruction falls into the fragment supported by modern datalog solvers, we found *Soufflé* more performant than using the general-purpose solver *z3* in this context [3]. However, for reasoning about more involved properties, the expressiveness of *z3* is required, as we will illustrate in Section 4.4.2.

4.4.2 Case Study: SafeMath Library

As a case study for functional correctness and assertion checking we chose *Solidity*'s `SafeMath` library [saf19], a library implementing proper exception behavior for standard arithmetic operations. This particularly encompasses exceptions in case of overflows, underflows, and division or modulo by 0. The `SafeMath` library is special in that it is not deployed as an own contract on the blockchain, but its functions get inlined during the compilation of a contract that uses them[4]. This specific behavior makes it particularly interesting to analyze the individual library functions as their concrete implementations may vary with changes in the compiler.

[3] *z3* implements a standard datalog engine which is restricted to work with predicates over finite domains. This constraint ensures that the `smt-lib`-expressible Horn clauses do not leave the datalog-solvable fragment. *Soufflé* overcomes this restriction in favor of a more liberal characterization of the solvable fragment, which could also be integrated into the *HoRSt* language - allowing for compilation to *Soufflé* from this fragment.

[4] In *Solidity*, one always needs to provide definitions of the (library) contracts one is interacting with. In case that a library is only containing pure internal functions, the *Solidity* compiler inlines this functions instead of compiling them to DELEGATECALL call instructions to an address at which the user specified the library to reside.

Functional Correctness. For our case study we compiled the functions of the `SafeMath` library with a recent stable *Solidity* compiler version (0.5.0) and verified that they expose the desired behavior. We showed that all functions 1) cannot return successfully in the problematic corner cases. 2) can return successfully with the correct result in the absence of corner cases. 3) if halting successfully in the absence of corner cases, they can return nothing but the correct result. As these properties require to precisely relate different input values over the execution (e.g., requiring that the sum of two input values exceeds 2^{256}), we needed to slightly adapt our analysis by adding a representation of the initial input (as word array) to the MState and the Halt predicates. This array is accessed by the CALLDATALOAD operation which fetches the input data. Additionally, we need to model return values by an own predicate. For more details, we refer the reader to Appendix C.3.3. *eThor* manages to prove the corresponding functional properties for each of the five functions within milliseconds, showcasing the tool's efficiency. Note that verifying meaningful functional correctness properties, like in this case study, requires to universally quantify over potential inputs, hence making an analysis with a datalog engine (such as *Soufflé*), which requires to explicitly list finite initial relations, infeasible.

Static Assertion Checking. The following code snippet shows the division function of the `SafeMath` library:

```
1 function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns
      (uint256) {
2     require(b > 0, errorMessage);
3     uint256 c = a / b;
4     // assert(a == b * c + a % b); // There is no case in which this doesn't hold
5     return c; }
```

It testifies that the function used to contain an assertion which was deemed to be unnecessary and hence removed (probably to save gas). We reinserted this assertion and indeed could prove that the dynamic assertion check is superfluous as it can never be violated.

4.4.3 Large-scale Evaluation

We performed a series of experiments to assess the overall performance of our tool. In particular, we systematically evaluated *eThor*'s correctness and precision on the official EVM test suite and additionally conducted a large scale analysis for the single-entrancy property, comparing *eThor* with the ZEUS [KGDS18a] static analyzer, using the real-world data set introduced with the latter[5].

Automated Testing. For a principled correctness assessment, we evaluated *eThor* against the virtual machine test cases provided by the Ethereum Foundation[6]. Being formulated as pre- and postconditions, these test cases fall in the class of properties characterized in Section 4.2.5, and we could automatically translate them into queries in *HoRSt*. The test suite defines 609 test cases,

[5] We compare with [KGDS18a] as we found it the only (claimed) sound tool to support a property comparable to single-entrancy. [TDDC+18] only supports a pattern (NW) which the authors claim to be different from reentrancy. [LWZ+19] utilizes a similar pattern.

[6] `https://github.com/ethereum/tests/`

604 of which specify properties relevant for a single contract setting (see Appendix C.3.4 for details). Using a 1 second timeout, we were able to solve 85% (513) of the test cases precisely with a termination rate of 99% (597).

Reentrancy. For the call unreachability property described in Definition 10, we evaluated *eThor* against the set of real-world contracts presented in [KGDS18a]. The authors extracted 22493 contracts from the Ethereum blockchain over a period of three months and (after deduplication) made available a list of 1524 contract addresses. Due to various problems of this data set (as described in [ext20]), sanitization leaves us with 720 distinct bytecodes, out of which we label 100 contracts to be trivially non-reentrant (due to the absence of possibly reentering instructions) and 2 were out of the analysis scope (containing at least one DELEGATECALL or CALLCODE instruction) and hence immediately classified to be potentially vulnerable. We make the sanitized benchmark available to the community, including bytecode and sources (where available) [ext20]. For 13 contracts, we failed to reconstruct the control-flow graph, leaving us with 605 distinct contracts to run our experiments on.

We ran three different experiments for evaluating *eThor*'s performance for the single-entrancy property, performing no folding, linear folding, and exhaustive Horn clause folding. This experimental setup aims not only to conduct a comparison with ZEUS, but also to showcase how *eThor*'s modular structure facilitates its performance in that *eThor* can flexibly benefit from different optimization techniques of the *HoRSt* compiler. In comparison with ZEUS, we take into account the combined result of the three different experiments (the contracts solvable using any of the applied foldings). For the exhaustive folding, we omitted instances where the time for `smt-lib` generation exceeded 15 minutes.[7] All of the experiments were conducted on a virtual machine with 64 Cores at 2 GHz and 128 GiB of RAM. At most 64 queries were executed at once, each with a 10 minutes timeout. Combining the different experiments we obtained results for all but 28 contracts.

We compared the results with [KGDS18a]. Because of the existing soundness concerns regarding [KGDS18a] (as discussed in Section 3.4 and also reported in the literature [TS+18]), we manually reassessed the ground truth for all contracts that were labeled insecure by at least one of the tools. Since this is a challenging and time consuming task, especially in the case that no *Solidity* source code is available, we excluded all contracts with more than 6000 bytecodes for which we were not able to obtain the source code, which leaves us with 709 contracts for which we assessed the ground truth.

Surprisingly, we found numerous contracts labeled non-reentrant by [KGDS18a] which, if analyzed in a single contract setting, definitely were reentrant according to the definition of reentrancy given in Definition 5 and also according to the informal definition provided in [KGDS18a] itself[8]. We assume this to be an artifact of [KGDS18a]'s syntactical treatment of the `call` directive on the *Solidity* level, which is, however, insufficient to catch all possible reentrancies. As the work excludes reentrancies introduced by the `send` directive (even though this is officially considered

[7] This timeout was chosen since it yielded a termination rate of $> 95\%$.

[8] [KGDS18a] gives the following informal definition: 'A function is reentrant if it can be interrupted while in the midst of its execution, and safely re-invoked even before its previous invocations complete execution.'

Measure	Definition	*eThor*	[KGDS18a]
termination	$terminated/total$	94.3	98.1
sensitivity	$tp/(tp+fn)$	100	11.4
specificity	$tn/(tn+fp)$	80.4	99.8
F-measure	$2*(spec*sens/(spec+sens))$	89.1	20.4

Table 4.1: Performance comparison of *eThor* and ZEUS [KGDS18a]. *total/terminated* denotes the number of contracts in the data set/the number of contracts the respective tool terminated on. *tp/fp* denotes the number of true/false positives and *tn/fn* the true/false negatives.

potentially insecure [Eth19]), for the sake of better comparability, we slightly updated our abstract semantics to account for calls that can be deemed secure following the same argument (namely that a small gas budget prevents reentrancy). We compare *eThor* against [KGDS18a] on our manually established ground truth. The results are summarized in Table 4.1.

For achieving a termination rate comparable to [KGDS18a] (94.3% vs. 98.1%), we needed to run our tool with a higher timeout (10 min. query timeout vs 1 min. contract time out for ZEUS). This difference can be explained by the fact that our analysis works on little structured bytecode in contrast to the simplified high-level representation used by [KGDS18a]. Additional overhead needs to be attributed to the usage of sound abstractions on the bytecode level as well as to our different experimental setup that did not allow for the same amount of parallelization. The soundness claim of [KGDS18a] is challenged by the experimentally assessed sensitivity of only 11.4%. One possible explanation for this low value, which deviates from the numbers reported in [KGDS18a] on the same data set, is that the intuition guiding the manual investigation performed by [KGDS18a] departed from the notion of single-entrancy and the intuitive definition given by the authors. This highlights the importance of formalizing not only the analysis technique but also the security properties to be verified. When interpreting the high specificity of [KGDS18a] (almost 100%) one should consider that ZEUS labels only 25 contracts vulnerable in total, out of which one is a false positive. Given that the data set is biased towards safe contracts (513 safe as opposed to 196 unsafe ones), a high specificity can be the result of a tool's tendency to label contracts erroneously secure. Due to the proven soundness, for *eThor* such behavior is excluded by design. This overall advantage of *eThor* over ZEUS in terms of accuracy is reflected by *eThor*'s F-measure of 89.1% as opposed to 20.4% for ZEUS.

Horn Clause Folding. Our experimental evaluation shows that, while both forms of Horn clause folding improve the termination rate, the results of the different foldings are not directly comparable. This is illustrated by Fig. 4.9 which plots the (lowest) termination times for those queries that terminated within 200 seconds during the large-scale experiment. The different colors indicate the kind of optimization (no/linear/exhaustive folding) that was fastest to solve the corresponding query. The distribution of the dots shows that in the range of low query times (indicating structured contracts), exhaustive folding (depicted in blue) dominates. However, for longer query times, the linear folding (depicted in green) often shows a better performance. One possible explanation is that for more complex contracts, the blow-up in rules created by the exhaustive folding exceeds the benefits of eliminating more predicates. Interestingly, for few

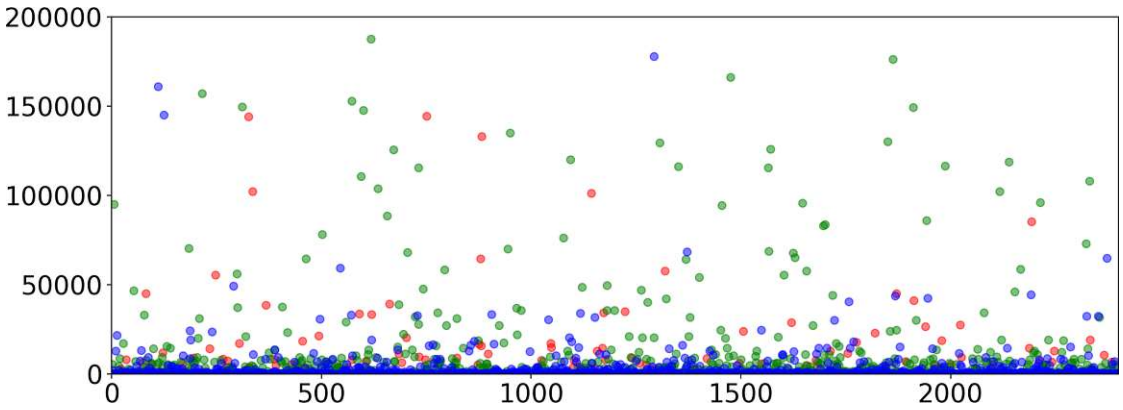

Figure 4.9: Query runtimes in ms for the combined approach itemized by queries. A red/green/blue dot denotes a query solved fastest with no/linear/exhaustive folding.

instances, even applying no folding at all (indicated in red) led to the fastest termination. We can only explain this behavior by special heuristics used inside *z3* that helped these particular cases. This shows the lacking predictability of *z3* and thereby motivates the necessity of high-level tools like *HoRSt* that allow users to easily combine different optimizations in order to obtain results reliably.

4.5 Discussion

As discussed in Chapter 3, in the last years, there have been plenty of works on the automatic analysis of Ethereum smart contractsspanning dynamic, as well as static analysis approaches. However, due to their security-critical nature, and since smart contracts once published on the blockchain are immutable, there is a particular need for *static and sound* analysis approaches.

Still, as detailed out in Section 3.4, so far all approaches to developing automated static analyzers that aim at providing soundness guarantees foundered on the particular challenges that the Ethereum environment poses to the performant verification of smart contracts.

For avoiding the pitfalls leading to unsoundness in the presented works, *eThor* follows a principled design approach: Starting from the formal EVM semantics defined in Chapter 2, it formulates an abstract semantics in the specification language *HoRSt* which is proven sound with respect to the concrete semantics, hence covering all particularities of the EVM bytecode language. Based on this abstract semantic specification, a streamlined compilation process creates an SMT encoding which is again systematically tested for soundness against the official test suite to minimize the effect of implementation bugs. The challenge of sound control flow reconstruction is solved by basing a corresponding preanalysis on a proper relaxation of the provably sound abstract semantics in the *Soufflé* format, ensuring that the original soundness guarantees are inherited[9]. For a more robust development, it is planned to also streamline this process in the future by

[9]The soundness of the control flow reconstruction is formally proven in [Gri21].

making the *HoRSt* compiler support *Soufflé* as an additional output format for a restricted Horn clause fragment. For providing end-to-end guarantees of the resulting static analyzer, we do not only ensure the soundness of the core analysis by proofs and testing but also give provably sound approximations for relevant formalized semantic security properties suitable for encoding in the analysis framework. One should mention that *eThor*'s soundness guarantees only hold in the absence of bugs in the implementations and the proofs. More precisely, this means that we need to assume that

1. The Ethereum clients implement the semantics as defined in Chapter 2.
2. *eThor* implements the Horn-clause-based abstraction as used in the soundness proof.
3. *eThor* implements all pre-processing steps correctly.
4. The underlying SMT solver is sound.
5. The (paper) proofs are correct.

With the described tool generation and testing pipeline, we try to mitigate the effects of potential bugs in implementations and proofs. In the future, we plan to add another assurance layer by mechanizing the manual soundness proofs.

4.6 Conclusion

We presented *eThor*, the first automated tool implementing a sound static analysis technique for EVM bytecode, showing how to abstract the semantics of EVM bytecode into a set of Horn clauses and how to express security as well as functional properties in terms of reachability queries, which are solved using *z3* . In order to ensure the long-term maintenance of the static analyzer and facilitate future refinements, we designed *HoRSt*, a development framework for Horn-clause-based static analysis tools, which given a high-level specification of Horn clauses automatically generates an optimized implementation in the `smt-lib` format. We successfully evaluated *eThor* against the official Ethereum test suite to gain further confidence in our implementation and conducted a large-scale evaluation, demonstrating the practicality of our approach. Within a large-scale experiment, we compared *eThor* to the state-of-the-art analysis tool ZEUS, demonstrating that *eThor* surpasses ZEUS in terms of overall performance (as quantified by the F-measure).

This work opens up several interesting research directions. For instance, we plan to extend our analysis as well as *HoRSt* to relational properties, since some interesting security properties, such as those presented in Section 2.4, for smart contracts can be defined in terms of 2-safety properties. Furthermore, we intend to further refine the analysis in order to enhance its precision, e.g., by extending the approach to a multi-contract setting, introducing abstractions for calls that approximate the account's persistent storage and local memory after calling more accurately. Furthermore, we plan to extend the scope of *HoRSt* significantly. First, we intend to make the specification of the static analysis accessible to proof assistants in order to mechanize soundness proofs. Furthermore, we intend to explore the automated generation of static analysis patterns

from the specification of the concrete semantics in order to further reduce the domain knowledge required in the design of static analyzers.

CHAPTER 5

Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability

Abstract

The tremendous growth in cryptocurrency usage is exposing the inherent scalability issues with permissionless blockchain technology. *Payment-channel networks* (PCNs) have emerged as the most widely deployed solution to mitigate the scalability issues, allowing the bulk of payments between two users to be carried out off-chain. Unfortunately, as reported in the literature and further demonstrated in this chapter, current PCNs do not provide meaningful security and privacy guarantees [GM17, MMSK+17].

In this chapter, we study and design secure and privacy-preserving PCNs. We start with a security analysis of existing PCNs, reporting a new attack that applies to all major PCNs, including the Lightning Network and allows an attacker to steal the fees from honest intermediaries in the same payment path. We then formally define anonymous multi-hop locks (AMHLs), a novel cryptographic primitive that serves as a cornerstone for the design of secure and privacy-preserving PCNs. We present several provably secure cryptographic instantiations that make AMHLs compatible with the vast majority of cryptocurrencies. In particular, we show that (linear) homomorphic one-way functions suffice to construct AMHLs for PCNs supporting a scripting language (e.g., Ethereum). We also propose a construction based on ECDSA signatures that *does not require scripts*, thus solving a prominent open problem in the field.

AMHLs constitute a generic primitive whose usefulness goes beyond multi-hop payments in a single PCN, and we show how to realize atomic swaps and interoperable PCNs from this primitive. Finally, our performance evaluation on a commodity machine finds that AMHL operations can be performed in less than 100 milliseconds and require less than 500 bytes of communication

overhead, even in the worst case. In fact, after acknowledging our attack, the Lightning Network developers have implemented our ECDSA-based AMHLs into their PCN. This demonstrates the practicality of our approach and its impact on the security, privacy, interoperability, and scalability of today's cryptocurrencies.

This chapter presents the result of a collaboration with Giulio Malavolta, Pedro Moreno Sanchez, Aniket Kate, and Matteo Maffei and was published at the Network and Distributed System Security Symposium (NDSS'19) under the title 'Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability' [MMSS+19]. I am responsible for the proof of the impossibility result that shows the inevitability of wormhole attacks in multi-hop payments with only two rounds of communication. Further, I contributed to the formalization of the newly introduced primitive and its security notion, and to the security proofs in the UC framework. Pedro Moreno Sanchez discovered the wormhole attack and developed together with Giulio Malavolta the ECDSA based construction and conducted the practical evaluation. Giulio Malavolta lead the development of the security proofs in the UC framework and the formalization of the primitive and its security notion. The accompanying appendix contains the detailed constructions and proofs.

5.1 Introduction

Cryptocurrencies are growing in popularity and are playing an increasing role in the worldwide financial ecosystem. In fact, the number of Bitcoin transactions grew by approximately 30% in 2017, reaching a peak of more than $420,000$ transactions per day in December 2017 [bit]. This striking increase in demand has given rise to scalability issues [CDE+16], which go well beyond the rapidly increasing size of the blockchain. For instance, the permissionless nature of the consensus algorithm used in Bitcoin today limits the transaction rate to tens of transactions per second, whereas other payment networks such as Visa support peaks of up to 47,000 transactions per second[vis].

Among the various proposals to solve the scalability issue [PD, DW15, LNZ+16, DRO], *payment-channels* have emerged as the most widely deployed solution in practice. In a nutshell, two users open a payment channel by committing a single transaction to the blockchain, which locks their bitcoins in a deposit secured by a Bitcoin (smart) contract. These users can then perform several payments between each other without the need for additional blockchain transactions by simply locally agreeing on the new deposit balance. A transaction is required only at the end in order to close the payment channel and unlock the final balances of the two parties, thereby drastically reducing the transaction load on the blockchain. Further research has proposed the concept of *payment-channel network* [PD] (PCN), where two users not sharing a payment channel can still pay each other using a path of open channels between them. Unfortunately, current PCNs fall short of providing adequate security, privacy, and interoperability guarantees.

5.1.1 State of the Art in PCNs

Several practical deployments of PCNs exist today [lnd, lig18, ecl] based on a common reference description for the Lightning Network (LN) [ln-b]. Unfortunately, this proposal is neither privacy-

preserving, as shown in recent works [GM17, MMSK+17], nor secure, which stays in contrast to what until now was commonly believed, as we show in this work. In fact, we present a new attack, the wormhole attack, which applies not only to the LN, the most widely deployed PCN, but also other PCNs based on the same cryptographic lock mechanism, such as the Raiden Network [rai].

PCNs have attracted plenty of attention also from academia. Malavolta et al. [MMSK+17] proposed a secure and privacy-preserving protocol for multi-hop payments. However, this solution is expensive as it requires exchanging a non-trivial amount of data (i.e., around 5 MB) between the users in the payment path, and it also hinders interoperability as it requires the Hash Time-Lock Contract (HTLC) supported in the cryptocurrency.

Green and Miers presented BOLT, a hub-based privacy-preserving payment for PCNs [GM17]. BOLT requires cryptographic primitives only available in Zcash, and it cannot be seamlessly deployed in Bitcoin. Moreover, this approach is limited to paths with a single intermediary, and the extension to support paths of arbitrary length remains an open problem.

The rest of the existing PCN proposals suffer from similar drawbacks. Apart from not formalizing provable privacy guarantees, they are restricted to a setting with a trusted execution environment [LNE+18] or with a Turing complete scripting language [DEFM17, DFH18, KG17, MBKM19] so that they cannot seamlessly work with prominent cryptocurrencies today (except for Ethereum).

Poelstra introduced the notion of scriptless scripts, a modified version of a digital signature scheme so that a signature can only be created when faithfully fulfilling a cryptographic condition [Poeb]. The resulting signature is verifiable following the unmodified digital signature scheme. When applied to script-based systems like Bitcoin or Ethereum, they are accompanied by core scripts (e.g., script to verify the signature itself). This approach reduces the space required for cryptographic operations in the script, saving thus invaluable bytes on the blockchain. Moreover, it improves upon the fungibility of the cryptocurrency as transactions from payment channels no longer require a script different from other payments.

Although interesting, current proposals [Poeb] lack formal security and privacy treatment and are based only on the Schnorr signature scheme, thus being incompatible with major cryptocurrencies like Bitcoin. Although there exist early proposals for Schnorr adoption in Bitcoin [Wui], it is unclear whether they will be realized.

In summary, existing proposals are neither generically applicable nor interoperable since they rely on specific features (e.g., contracts) of individual cryptocurrencies or trusted hardware. Furthermore, there seems to be a gap between the secure realization of PCNs and what is developed in practice, as we demonstrate with our attack, which affects virtually all deployed PCNs.

5.1.2 Our Contributions

In this work, we contribute to the rigorous understanding of PCNs and present the first interoperable, secure, and privacy-preserving cryptographic construction for multi-hop locks (AMHLs). Specifically,

- We analyze the security of existing PCNs, reporting a new attack (the *wormhole attack*) which allows dishonest users to steal the payment fees from honest users along the path. This attack applies to the LN, as well as any decentralized PCN (following the definition in [MMSK+17]) where the sender does not know in advance the intermediate users along the path to the receiver. We communicated the attack to the LN developers, who acknowledged the issue.
- In order to construct secure and privacy-preserving PCNs, we introduce a novel cryptographic primitive called anonymous multi-hop lock (AMHL). We model the security of such a primitive in the UC framework [Can01] to inherit the underlying composability guarantees. Then we show that AMHLs can be generically combined with any blockchain to construct a fully-fledged PCN.
- As a theoretical insight emerging from the wormhole attack, we establish a lower bound on the communication complexity of secure PCNs (Section 5.3) that follow the definition from [MMSK+17]: Specifically, we show that an extra round of communication to determine the path is necessary to have a secure transaction.
- We show how to realize AMHLs in different settings. In particular, we demonstrate that (linearly) homomorphic operations suffice to build any script-based AMHL. Furthermore, we show how to realize AMHLs in a scriptless setting. This approach is of special interest because it reduces the transaction size and, consequently, the blockchain load. We give a concrete construction based on the ECDSA signature scheme, solving a prominent problem in the literature [Poeb]. This makes AMHLs compatible with the vast majority of cryptocurrencies (including Bitcoin and Ethereum). In fact, AMHLs have been implemented and tested in the LN [Froa, Frob].
- We implemented our cryptographic constructions and show that they require at most 60 milliseconds to be computed and communication overhead of fewer than 500 bytes in the worst case. These results demonstrate that AMHLs are practical and ready to be deployed. In fact, AMHLs can be leveraged to design atomic swaps and interoperable (cross-currency) PCNs.

Organization. Section 5.2 shows the background on PCNs. Section 5.3 describes the wormhole attack. Section 5.4 formally defines AMHLs. Section 5.5 contains our protocols for AMHLs and Section 5.6 analyzes their performance. Section 5.7 describes applications for AMHLs. Section 5.8 discusses the related work and Section 5.9 concludes this chapter.

5.2 Context: Payment Channel Networks

5.2.1 Payment Channels

A payment channel allows two users to exchange bitcoin without committing every single payment to the Bitcoin blockchain. For that, users first publish an on-chain transaction to deposit bitcoin into a multi-signature address controlled by both users. Such deposit also guarantees that all bitcoin are refunded at a possibly different but mutually agreed time if the channel expires. Users can then perform off-chain payments by adjusting the distribution of the deposit (that we

will refer to as *balance*) in favor of the payee. When no more off-chain payments are needed (or the capacity of the payment channel is exhausted), the payment channel is closed with a *closing* transaction included in the blockchain. This transaction sends the deposited bitcoin to each user according to the most recent balance in the payment channel. We refer the reader to [PD, DW15, MMSH16, DRO] for further details.

5.2.2 A Payment Channel Network (PCN)

A PCN can be represented as a directed graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, where the set $\mathbb{V}$ of vertices represents the Bitcoin accounts, and the set $\mathbb{E}$ of weighted edges represents the payment channels. Every vertex $U \in \mathbb{V}$ has associated a non-negative number that denotes the fee that it charges for forwarding payments. The weight on a directed edge $(U_1, U_2) \in \mathbb{E}$ denotes the amount of remaining bitcoin that U_1 can pay to U_2.

A PCN is used to perform off-chain payments between two users with no direct payment channel between them but rather connected by a path of open payment channels. For that, assume that S wants to pay α bitcoin to R, which is reachable through a path of the form $S \to U_1 \to \ldots \to U_n \to R$. For their payment to be successful, every link must have a capacity $\gamma_i \geq \alpha'_i$, where $\alpha'_i = \alpha - \sum_{j=1}^{i-1} \mathit{fee}(U_j)$ (i.e., the initial payment value minus the fees charged by intermediate users in the path). If the payment is successful, edges from S to R are decreased by α'_i. Importantly, to ensure that R receives exactly α bitcoin, S must start the payment with a value $\alpha^* = \alpha + \sum_{j=1}^{n} \mathit{fee}(U_j)$. We refer the reader to [PD, MMSK$^+$17, MMSH16, GM17] for further details.

The concepts of payment channels and PCNs have already attracted considerable attention from academia [HAB$^+$17, LEPS16, DW15, GM17, MMSK$^+$17, MMSH16, MBKM19]. In practice, the Lightning Network (LN) [PD, ln-b] has emerged as the most prominent example. Currently, there exist several independent implementations of the LN for Bitcoin [lnd, lig18, ecl]. Moreover, the LN is also considered as a scalability solution in other blockchain-based payment systems such as Ethereum [rai].

5.2.3 Multi-hop Payments Atomicity

A fundamental property for multi-hop payments is *atomicity*: Either the capacity of all channels in the path is updated, or none of the channels is changed. Partial updates can lead to coin losses for the users on the path. For instance, a user could pay a certain amount of bitcoin to the next user in the path but never receive the corresponding bitcoin from the previous neighbor. The LN tackles this challenge by relying on a smart contract called *Hash Time-Lock Contract* (HTLC) [PD]. This contract locks x bitcoin that can be released only if the contract's condition is fulfilled. The contract relies on a collision-resistant hash function H and it is defined in terms of a hash value $y := H(R)$, where R is chosen uniformly at random, the amount of bitcoin x, and a timeout t, as follows: (i) If Bob produces the condition R^* such that $H(R^*) = y$ before t days, Alice pays Bob x bitcoin; (ii) If t days elapse, Alice gets back x bitcoin.

Fig. 5.1 shows an example of the use of HTLC in a payment. For simplicity, we assume that every user charges a fee of one bitcoin, and the payment amount is 10 bitcoin. In this payment,

Edward first sets up the payment by creating a random value R and sending $H(R)$ to Alice. Then, the *commitment phase* starts with Alice. She first sets on hold 13 bitcoin, and then successively, every intermediate user sets on hold the received amount minus his/her own fee. After Dave sets 10 coins on hold with Edward, the latter knows that the corresponding payment amount is on hold at each channel and he can start the *releasing phase* (depicted in green). For that, he reveals the value R to Dave, allowing him to fulfill the HTLC contract and settle the new capacity at the payment channel. The value R is then passed back along the path, allowing the settlement of the payment channels.

Privacy Issues in PCNs. Recent works [MMSK+17, GM17] show that the current use of HTLC leaks a common identifier along the payment path (i.e., the condition $H(R)$) that can be used by an adversary to tell who pays to whom. Current solutions to this privacy issue are expensive in terms of computation and communication [MMSK+17] or incompatible with major cryptocurrencies [GM17]. This calls for an in-depth study of this cryptographic tool.

5.3 Wormhole Attack in Existing PCNs

In a nutshell, the wormhole attack allows two colliding users on a payment path to exclude intermediate users from participating in the successful completion of a payment, thereby stealing the payment fees which were intended for honest path nodes.

In more detail, assume a payment path $(U_0, \ldots, U_i, \ldots, U_j, \ldots, U_n)$ used by U_0 to pay an amount $\alpha + \sum_k \gamma_k$ to U_n, where $\gamma_k = fee(U_k)$ denotes the fee charged by the intermediate user U_k as a reward for enabling the payment. Further assume that U_i and U_j are two adversarial users that may deviate from the protocol if some economic benefit is at stake. The adversarial strategy is as follows.

In the commitment phase, every user behaves honestly. This, in particular, implies that every honest user has locked a certain amount of coins in the hope of getting rewarded for this. In the releasing phase, honest users $U_{j+1}, \ldots, U_n$ correctly fulfill their HTLC contracts and settle the balances and rewards in their corresponding payment channels.

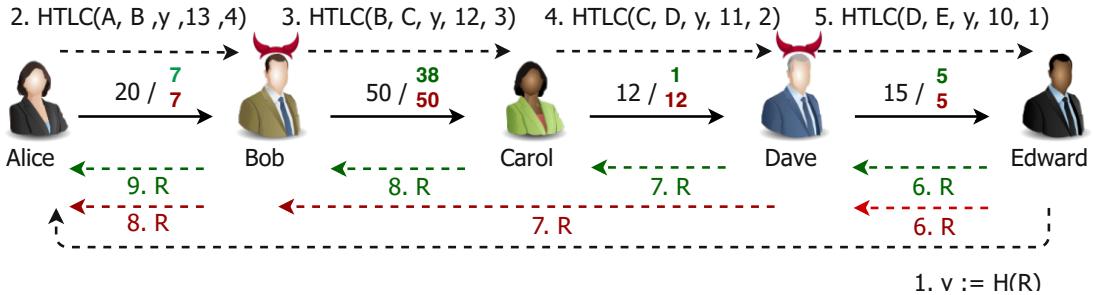

Figure 5.1: Payment (with and without wormhole attack) from Alice to Edward for value 10 using HTLC contract. The honest (attacked) releasing phase is depicted in green (red). Non-bold (bold) numbers show the capacity of payment channels before (after) the payment. We assume a common fee of 1 coin.

The user U_j behaves honestly with U_{j+1} effectively settling the balance in their payment channel. On the other hand, U_j waits until the timeout set in the HTLC with U_{j-1} is about to expire and then agrees with U_{j-1} to cancel the HTLC and set the balance in their payment channel back to the last agreed one. Note that from U_{j-1}'s point of view, this is a legitimate situation (e.g., there might not be enough coins in a payment channel at some user after U_j and the payment had to be canceled). Moreover, the channel between U_{j-1} and U_j does not need to be closed, and it is just rolled back to a previous balance, a feature present in the Lightning Network.

As U_{j-1} believes that the payment did not go through, she also cancels the HTLC with U_{j-2}, who in turn cancels the HTLC with U_{j-3} and so on. This process continues until U_i is approached by U_{i+1}. Here, U_i cancels the HTLC with U_{i+1}. However, U_i gets the releasing condition R from U_j and can use it to fulfill the HTLC with U_{i-1} and therefore settle the new balance in that payment channel. Therefore, from the point of view of users $U_1, \ldots, U_{i-1}$, the payment has been successfully carried out.

An illustrative example of this attack is shown in Fig. 5.1 with the attacked releasing phase depicted in red.

Discussion. An adversary controlling users U_i and U_j in a payment path that carries out the attack described in this section gets an overall benefit of $\sum_{k=i+1}^{j} \gamma_k$ bitcoins instead of only $\gamma_i + \gamma_j$ bitcoins in the case he behaves honestly. We make several observations here. First, the impact of this attack grows with the number of intermediate users between U_i and U_j as well as the number of payments that take both U_i and U_j in their path. While the Lightning Network is at its infancy, other well-established networks such as Ripple use paths with multiple intermediaries. For instance, in the Ripple network, more than 27% of the payments use more than two intermediaries [MMS+18]. Actually, paths with three intermediaries (e.g., sender $\rightarrow$ bank $\rightarrow$ currency-exchange $\rightarrow$ bank $\rightarrow$ receiver) are essential for currency exchanges, a key use case in LN itself [ln-a]. When the LN grows to the scale of the Internet, routes may consist of several intermediaries as on the Internet today. Given this evidence, we expect long paths in the LN.

Second, honest intermediate users cannot trivially distinguish the situation in which they are under attack from the situation where the payment is simply unsuccessful (e.g., there are not enough coins in one of the channels or one of the users is offline). In both cases, the view for the honest users is that the timeout established in the HTLC is reached, the payment failed, and they get their initially committed coins reimbursed. In short, the wormhole attack allows an adversary to steal the fees from intermediate honest users without leaving an inculpatory trace to them.

Third, fees are the main incentive for intermediary users. The wormhole attack takes away this crucial benefit. In fact, this attack not only makes honest users lose their fees but also incur collateral costs: Coins locked for the payment under attack cannot be used for another (possibly successful) payment simultaneously.

Responsible Disclosure. We notified this attack to the LN developers and they have acknowledged this issue. Additionally, they have implemented our ECDSA-based construction (see Sec-

tion 5.5.4) and tested it for its integration in the LN, having thereby a fix for the wormhole attack and leveraging its privacy and practical benefits [Froa, Frob].

(In)evitability of the Wormhole Attack. The wormhole attack is not restricted to the LN but generally applies to PCNs with multi-hop payments that involve only two rounds of communication. We assume a communication round to consist of traversing the payment path once, either forth (e.g., for setting up the payment) or back (e.g., for releasing the coins). Additionally, we assume that in PCNs the communication between nodes is restricted to their direct neighbors, so in particular, there is no broadcast.[1] Consequently, using two rounds of communication for a payment implies that the payment is not preceded by a routing phase in which path-specific information is sent to nodes in the path.

Under these assumptions, we state the lower bound informally in Theorem 1 and defer the formal theorem and the proof to Appendix D.1.

Theorem 1 (Informal)**.** *For all two-round (without broadcast channels) multi-hop payment protocols, there exists a path prone to the wormhole attack.*

In this work, we show that adding an additional round of communication suffices to overcome this impossibility result.[2] In particular, with one additional round of communication, the sender of a payment can communicate path-specific secret information to the intermediate nodes. This information can then be used to make the release keys unforgeable for an attacker. The cryptographic protocols we introduce in the remainder of this chapter adopt this approach.

5.4 Definition

Here we introduce a new cryptographic primitive called anonymous multi-hop lock (AMHL). This primitive generalizes the locking mechanism used for payments in state-of-the-art PCNs such as the LN. In Section 5.7 we show that AMHL is the main cryptographic component required to construct fully-fledged PCNs. As motivated in the previous section, we model the primitive such that it allows for an initial setup phase where the first node of the path provides the other nodes with some secret (path-specific) state. Formally, an AMHL is defined with respect to a universe of users $\mathbb{U}$ and it is a five-tuple of PPT algorithms and protocols $\mathbb{L} = (\mathsf{KGen}, \mathsf{Setup}, \mathsf{Lock}, \mathsf{Rel}, \mathsf{Vf})$ defined as follows:

Definition 1. *An* AMHL $\mathbb{L} = (\mathsf{KGen}, \mathsf{Setup}, \mathsf{Lock}, \mathsf{Rel}, \mathsf{Vf})$ *consists of the following efficient algorithms:*

$\{(\mathsf{sk}_i, \mathsf{pk}), (\mathsf{sk}_j, \mathsf{pk})\} \leftarrow \langle \mathsf{KGen}_{U_i}(1^\lambda), \mathsf{KGen}_{U_j}(1^\lambda) \rangle$: *On input the security parameter* 1^λ *the key generation protocol returns a shared public key* pk *and a secret key* sk_i (sk_j*, respectively) to* U_i *and* U_j*.*

[1] This is the case in the setting of off-chain protocols where users not sharing a payment channel do not communicate with each other.

[2] A malicious sender can still bypass intermediate nodes, but he has no incentive as it implies stealing coins from himself.

$\{s_0^I, \ldots, (s_n^I, k_n)\} \leftarrow \langle \mathsf{Setup}_{U_0}(1^\lambda, U_1, \ldots, U_n), \mathsf{Setup}_{U_1}(1^\lambda), \ldots, \mathsf{Setup}_{U_n}(1^\lambda) \rangle$: *On input a vector of identities* $(U_1, \ldots, U_n)$ *and the security parameter* 1^λ*, the setup protocol returns, for* $i \in [0, n]$*, a state* s_i^I *to user* U_i*. The user* U_n *additionally receives a key* k_n*.*

$\{(\ell, s_i^R), (\ell, s_{i+1}^L)\} \leftarrow \langle \mathsf{Lock}_{U_i}(s_i^I, \mathsf{sk}_i, \mathsf{pk}), \mathsf{Lock}_{U_{i+1}}(s_{i+1}^I, \mathsf{sk}_{i+1}, \mathsf{pk}) \rangle$: *On input two initial states* s_i^I *and* s_{i+1}^I*, two secret keys* sk_i *and* sk_{i+1}*, and a public key* pk*, the locking protocol is executed between two users* (U_i, U_{i+1}) *and returns a lock* ℓ *and a right state* s_i^R *to* U_i *and the same lock* ℓ *and a left state* s_{i+1}^L *to* U_{i+1}*.*

$k' \leftarrow \mathsf{Rel}(k, (s^I, s^L, s^R))$: *On the input of an opening key* k *and a triple of states* (s^I, s^L, s^R)*, the release algorithm returns a new opening key* k'*.*

$\{0, 1\} \leftarrow \mathsf{Vf}(\ell, k)$: *On input a lock* ℓ *and a key* k *the verification algorithm returns a bit* $b \in \{0, 1\}$*.*

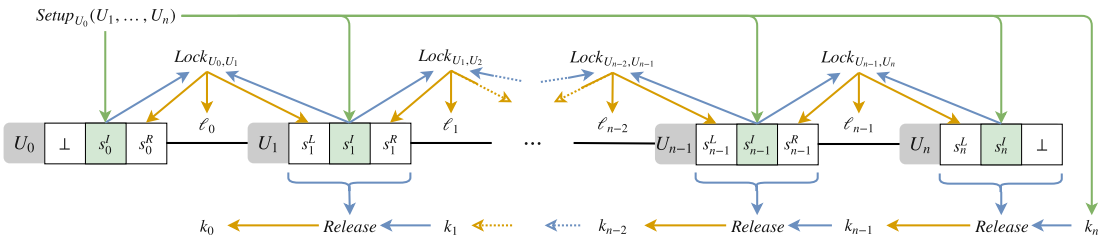

Figure 5.2: Usage of the AMHL primitive. It is assumed that links between the users on the path have been created upfront (using KGen) and that the resulting public and secret keys are implicitly given as argument to the corresponding executions of Lock. Otherwise, the inputs (outputs) to (from) the Lock protocol and the Rel algorithm are indicated by blue (orange) arrows.

Correctness. An AMHL is *correct* if the verification algorithm Vf always accepts an honestly generated lock-key pair. For a more detailed and formal correctness definition, we refer the reader to Appendix D.2.

Key Ideas. Fig. 5.2 illustrates the usage of the different protocols underlying the AMHL primitive. First, we assume an (interactive) KGen phase that emulates the opening of payment channels that compose the PCN.

In the setup phase (green arrows), the introduction of the initial state at each intermediate user is crucial for security and privacy. Intuitively, we can use this initial state as "rerandomization factor" to ensure that locks in the same path are unlinkable for the adversary.

Next, in the locking phase, each pair of users jointly executes the Lock protocol to generate a lock ℓ_i. The creation of this lock represents the commitment from U_i to perform an application-dependent action if a cryptographic problem is solved by U_{i+1}. In the case of LN, this operation represents the commitment of U_i to pay a certain amount of coins to U_{i+1} if U_{i+1} solves the cryptographic condition. Each user also learns some extra state s_i^R (resp. s_{i+1}^L) that will be

needed for releasing the lock later on. While these extra states are not present in the LN (i.e., every lock is based on the same cryptographic puzzle $H(R)$), they are crucial for security. They make the releasing of different locks in the path independent and thus ensure that a lock ℓ_i can only be released if ℓ_{i+1} has been released before.

Finally, after the entire path is locked, the receiver U_n can generate a key for releasing its left lock. Then, each intermediate node can derive a valid key for its left lock from a valid key for its right lock using the Rel algorithm. This last phase resembles the opening phase of the LN, where each pair of users settles the new balances for their deposit at each payment channel in the payment path.

5.4.1 Security and Privacy Definition

To model security and privacy in the presence of concurrent executions, we resort to the universal composability framework from Canetti [Can01]. We allow thereby the composition of AMHLs with other application-dependent protocols while maintaining security and privacy guarantees.

Attacker Model. We model the players in our protocol as interactive Turing machines that communicate with a trusted functionality $\mathcal{F}$ via secure and authenticated channels. We model the attacker $\mathcal{A}$ as a PPT machine that has access to an interface $\mathsf{corrupt}(\cdot)$ that takes as input a user identifier U and provides the attacker with the internal state of U. All the subsequent incoming and outgoing communication of U are then routed through $\mathcal{A}$. We consider the static corruption model. That is, the attacker is required to commit to the identifiers of the users he wishes to corrupt ahead of time.[3]

Communication Model. Communication happens through the secure message transmission functionality $\mathcal{F}_{\mathsf{smt}}$ that informs the attacker whenever some communication happens between two users and the attacker can delay the delivery of the message arbitrarily (for a concrete functionality see [Can01]). We also assume the existence of a functionality $\mathcal{F}_{\mathsf{anon}}$ (see [CL05] for an example), which provides user with an anonymous communication channel. In its simplest form, $\mathcal{F}_{\mathsf{anon}}$ is identical to $\mathcal{F}_{\mathsf{smt}}$, except that it omits the identifier of the sender from the message sent to the receiver. We assume a synchronous communication network, where the execution of the protocol happens in discrete rounds. The parties are always aware of the current round and if a message is created at round i, then it is delivered at the beginning of the $(i+1)$-th round. Our model assumes that computation is instantaneous. In the real world, this is justified by setting a maximum time bound for message transmission, which is known by all users. If no message is delivered by the expiration time, then the message is set to be $\perp$. We remark that such an assumption is standard in the literature [DEFM17] and for an example of the corresponding ideal functionality $\mathcal{F}_{\mathsf{syn}}$ we refer the reader to [Can01, KMTZ13].

Universal Composability. Let $\mathsf{EXEC}_{\tau,\mathcal{A},\mathcal{E}}$ be the ensemble of the outputs of the environment $\mathcal{E}$ when interacting with the attacker $\mathcal{A}$ and users running protocol τ (over the random coins of all the involved machines).

[3] Extending our protocol to support adaptive corruption queries is an interesting open problem.

Definition 2 (Universal Composability)**.** *A protocol τ UC-realizes an ideal functionality $\mathcal{F}$ if for any PPT adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for any environment $\mathcal{E}$ the ensembles* $\mathsf{EXEC}_{\tau,\mathcal{A},\mathcal{E}}$ *and* $\mathsf{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{E}}$ *are computationally indistinguishable.*

KeyGen(*sid*, U_j, $\{L, R\}$)

Upon invocation by U_i:
$\mathsf{send_s}$ $(sid, U_i, \{L, R\})$ to U_j
$\mathsf{receive_s}$ (sid, b) from U_j
if $b = \perp$ send $\perp$ to U_i and abort
if L insert (U_i, U_j) into $\mathcal{U}$ and $\mathsf{send_s}$ (sid, U_i, U_j) to U_i
if R insert (U_j, U_i) into $\mathcal{U}$ and $\mathsf{send_s}$ (sid, U_j, U_i) to U_i

Lock(*sid*, *lid*)

Upon invocation by U_i:
if *getStatus*$(lid) \neq \mathsf{Init}$ or *getLeft*$(lid) \neq U_i$ then abort
$\mathsf{send_s}$ $(sid, lid, \mathsf{Lock})$ to *getRight*(lid)
$\mathsf{receive_s}$ (sid, b) from *getRight*(lid)
if $b = \perp$ send $\perp$ to U_i and abort
updateStatus(lid, Lock)
$\mathsf{send_s}$ $(sid, lid, \mathsf{Lock})$ to U_i

GetStatus(*sid*, *lid*)

Upon invocation by U_i:
return $(sid, lid,$ *getStatus*$(lid))$ to U_i

Setup(*sid*, $U_0, \ldots, U_n$)

Upon invocation by U_0:
if $\forall i \in [0, n-1] : (U_i, U_{i+1}) \notin \mathcal{U}$ then abort
$\forall i \in [0, n-1]$: $lid_i \leftarrow\$ \{0,1\}^\lambda$
insert $(lid_0, U_0, U_1, \mathsf{Init}, lid_1), (lid_{n-1}, U_{n-1}, U_n, \mathsf{Init}, \perp)$ into $\mathcal{L}$
$\mathsf{send_{an}}$ $(sid, \perp, lid_0, \perp, U_1, \mathsf{Init})$ to U_0
$\mathsf{send_{an}}$ $(sid, lid_{n-1}, \perp, U_{n-1}, \perp, \mathsf{Init})$ to U_n
$\forall i \in [1, n-1]$: insert $(lid_i, U_i, U_{i+1}, \mathsf{Init}, lid_{i+1})$ into $\mathcal{L}$
$\mathsf{send_{an}}$ $(sid, lid_{i-1}, lid_i, U_{i-1}, U_{i+1}, \mathsf{Init})$ to U_i

Release(*sid*, *lid*)

Upon invocation by U_i:
if *getRight*$(lid) \neq U_i$ or *getStatus*$(lid) \neq \mathsf{Lock}$ or
  getStatus(*getNextLock*$(lid)) \neq \mathsf{Rel}$
  and *getNextLock*$(lid) \neq \perp$ then abort
updateStatus(lid, Rel)
$\mathsf{send_s}$ (sid, lid, Rel) to *getLeft*(lid)

Figure 5.3: Ideal functionality for cryptographic locks (AMHLs)

Ideal Functionality. We formally define the ideal world functionality $\mathcal{F}$ for AMHLs in the following. For a more modular treatment, our UC definition models only the cryptographic lock functionality, rather than aiming at a comprehensive characterization of PCNs. In Section 5.7 we show how one can construct a full PCN (e.g., as defined in [MMSK+17]) by composing this functionality with time locks, balance updates, and on-chain channel management. For ease of exposition, we assume that each pair of users establishes only a single link per direction. The model can be easily extended to handle the more generic case. $\mathcal{F}$ works in interaction with a universe of users $\mathbb{U}$ and initializes two empty lists $(\mathcal{U}, \mathcal{L}) := \emptyset$, which are used to track the users and the locks, respectively. The list $\mathcal{L}$ represents a set of lock chains. The entries are of the form $(lid_i, U_i, U_{i+1}, f, lid_{i+1})$ where lid_i is a lock identifier that is unique even among other lock chains in $\mathcal{L}$, U_i and U_{i+1} are the users connected by the lock, $f \in \{\mathsf{Init}, \mathsf{Lock}, \mathsf{Rel}\}$ is a flag that represents the status of the lock, and lid_{i+1} is the identifier of the next lock in the path. For sake of better readability, we define functions operating on $\mathcal{L}$ extracting lock-specific information given the lock's identifier, such as the lock's status (*getStatus*$(\cdot)$), the nodes it is connecting

(*getLeft*$(\cdot)$, *getRight*$(\cdot)$), and the next lock's identifier (*getNextLock*$(\cdot)$). In addition we define an update function *updateStatus*$(\cdot,\cdot)$ that changes the status of a lock to a new flag.

The interfaces of the functionality $\mathcal{F}$ are specified in Fig. 5.3. The KeyGen interface allows a user to establish a link with another user (specifying whether it wants to be the left or the right part of the link). The Setup interface allows a user U_0 to set up a path (starting from U_0) along previously established links. The Lock interface allows a user to create a lock with its right neighbor on a previously created path and the Release algorithm allows a user to release the lock with its left neighbor, in case that the user is either the receiver or its right lock has been released before. Finally, the GetStatus interface allows one to check the current status of a lock, i.e., whether it is initialized, locked, or released. Internally, the locks are assigned identifiers that are unique across all paths. We define the interfaces $\mathsf{send}_{\mathsf{s}}$ and $\mathsf{receive}_{\mathsf{s}}$ to exchange messages through the $\mathcal{F}_{\mathsf{smt}}$ functionality and the interface $\mathsf{send}_{\mathsf{an}}$ to send messages via $\mathcal{F}_{\mathsf{anon}}$.

5.4.2 Discussion

We discuss how the security and privacy notions of interest for AMHLs are captured by functionality $\mathcal{F}$.

Atomicity. Loosely speaking, atomicity means that every user in a path can release its left lock in case his right lock was already released. This is enforced by $\mathcal{F}$ as i) it is keeping track of the chain of locks and their current status in the list $\mathcal{L}$ and ii) the Release interface of $\mathcal{F}$ allows one to release a lock *lid* (changing the flag to Rel) if *lid* is locked and the follow-up lock (*getNextLock*(lid)) was already released.

Consistency. An AMHL is consistent if no attacker can release his left lock without its right lock being released before. This prevents scenarios where some AMHL is released before the receiver is reached and, more generically, the wormhole attack described in Section 5.3. To see why our ideal functionality models this property, observe that the Release interface allows a user to release the left lock only if the right lock has already been released or the user itself is the receiver. In this context, no wormhole attack is possible as intermediate nodes cannot be bypassed.

Relationship Anonymity. Relationship anonymity [BKM+13] requires that each intermediate node does not learn any information about the set of users in an AMHL beyond its direct neighbors. This property is satisfied by $\mathcal{F}$ as the lock identifiers are sampled at random, and during the locking phase, a user only learns the identifiers of its left and right lock as well as its left and right neighbor. We discuss this further in Appendix D.4.

5.5 Constructions

5.5.1 Cryptographic Building Blocks

Throughout this work, we denote by $1^\lambda \in \mathbb{N}^+$ the security parameter. Given a set S, we denote by $x \leftarrow\!\!\$\ S$ the sampling of an element uniformly at random from S, and we denote by $x \leftarrow A(in)$ the output of the algorithm A on input *in*. We denote by $\mathsf{min}(a,b)$ the function that takes as input two

integers and returns the smaller of the two. To favor readability, we omit session identifiers from the description of the protocols. In the following, we briefly recall the cryptographic building blocks of our schemes.

Homomorphic One-way Functions. A function $g : \mathcal{D} \to \mathcal{R}$ is one-way if, given a random element $x \in \mathcal{R}$, it is hard to compute a $y \in \mathcal{D}$ such that $g(y) = x$. We say that a function g is homomorphic if $\mathcal{D}$ and $\mathcal{R}$ define two abelian groups and for each pair $(a, b) \in \mathcal{D}^2$ it holds that $g(a \circ b) = g(a) \circ g(b)$, where $\circ$ denotes the group operation. Throughout this work we denote the corresponding arithmetic group additively.

Commitment Scheme. A commitment scheme COM consists of a commitment algorithm $(\mathsf{decom}, \mathsf{com}) \leftarrow \mathsf{Commit}(1^\lambda, m)$ and a verification algorithm $\{0,1\} \leftarrow \mathsf{V}_{\mathsf{com}}(\mathsf{com}, \mathsf{decom}, m)$. The commitment algorithm allows a prover to commit to a message m without revealing it. In a second phase, the prover can convince a verifier that the message m was indeed committed by showing the unveil information decom. The security of a commitment scheme is captured by the standard ideal functionality $\mathcal{F}_{\mathsf{com}}$ [Can01].

Non-interactive Zero-knowledge. Let R be an NP relation and let L be the set of positive instances, i.e., $L := \{x \mid \exists w \text{ s.t. } R(x, w) = 1\}$. A non-interactive zero-knowledge proof [BFM88] scheme NIZK consists of an efficient prover algorithm $\pi \leftarrow \mathsf{P}_{\mathsf{NIZK}}(w, x)$ and an efficient verifier $\{0,1\} \leftarrow \mathsf{V}_{\mathsf{NIZK}}(x, \pi)$. A NIZK scheme allows the prover to convince the verifier about the existence of a witness w for a certain statement x without revealing any additional information. The security of a NIZK scheme is modeled by the following ideal functionality $\mathcal{F}_{\mathsf{NIZK}}$: On input $(\mathsf{prove}, \mathit{sid}, x, w)$ by the prover, check if $R(x, w) = 1$ and send $(\mathsf{proof}, \mathit{sid}, x)$ to the verifier if this is the case.

Homomorphic Encryption. One of the building blocks of our work is the additive homomorphic encryption scheme $\mathsf{HE} := (\mathsf{KGen}_{\mathsf{HE}}, \mathsf{Enc}_{\mathsf{HE}}, \mathsf{Dec}_{\mathsf{HE}})$ from Paillier [Pai99]. The scheme supports homomorphic operation over the ciphertexts of the form $\mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, m) \cdot \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, m') = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, m + m')$. We assume that Paillier's encryption scheme satisfies the notion of ecCPA security, as defined in the work of Lindell [Lin17].

ECDSA Signatures. Let $\mathbb{G}$ be an elliptic curve group of order q with base point G and let $H : \{0,1\}^* \to \{0,1\}^{|q|}$ be a collision resistant hash function. The key generation algorithm $\mathsf{KGen}_{\mathsf{ECDSA}}(1^\lambda)$ samples a private key as a random value $x \leftarrow\!\!\$\, \mathbb{Z}_q$ and sets the corresponding public key as $Q := x \cdot G$. To sign a message m, the signing algorithm $\mathsf{Sig}_{\mathsf{ECDSA}}(\mathsf{sk}, m)$ samples some $k \leftarrow\!\!\$\, \mathbb{Z}_q$ and computes $e := H(m)$. Let $(r_x, r_y) := R = k \cdot G$, then the signing algorithm computes $r := r_x \bmod q$ and $s := \frac{e + rx}{k} \bmod q$. The signature consists of (r, s). The verification algorithm $\mathsf{Vf}_{\mathsf{ECDSA}}(\mathsf{pk}, \sigma, m)$ recomputes $e = H(m)$ and returns 1 if and only if $(x, y) = \frac{e}{s} \cdot G + \frac{r}{s} \cdot Q$ and $r = x \bmod q$. It is a well known fact that for every valid signature (r, s), also the pair $(r, -s)$ is a valid signature. To make the signature *strongly* unforgeable we augment the verification equation with a check that $s \leq \frac{q-1}{2}$. We assume the existence of an interactive protocol $\Pi^{\mathsf{ECDSA}}_{\mathsf{KGen}}$ executed between two users where the one receives (x_0, Q, sk), where sk is a Paillier secret key and $Q = x_0 \cdot x_1 \cdot G$, whereas the other obtains $(x_1, Q, \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, x_0))$, where pk is the corresponding Paillier public-key. For correctness, we require that the Paillier modulus

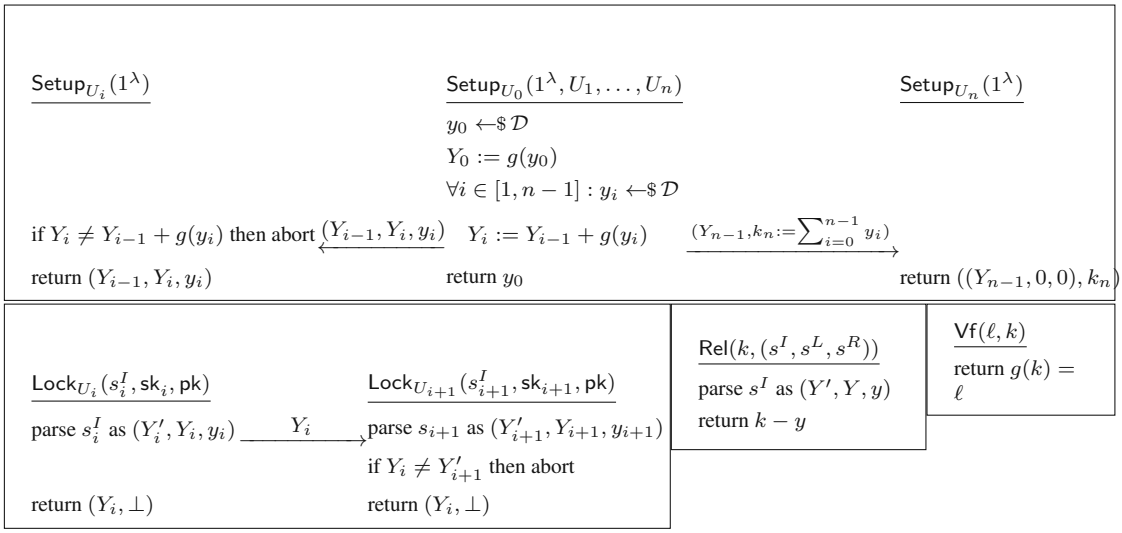

Figure 5.4: Algorithms and protocols for the generic construction

is $N = O(q^4)$. We assume that the parties have access to an ideal functionality $\mathcal{F}_{\mathsf{kgen}}^{\mathsf{ECDSA}}$ (refer to Appendix D.5 for a precise definition) that securely computes the tuples for both parties. An efficient protocol has been recently proposed by Lindell [Lin17].

Anonymous Communication. We assume an anonymous communication channel Π_{anon} available among users in the network, which is modeled by the ideal functionality $\mathcal{F}_{\mathsf{anon}}$. It anonymously delivers messages to users in the network (e.g., see [CL05]).

5.5.2 Generic Construction

An interesting question related to AMHLs is under which class of hard problems such a primitive exists. A generic construction using trapdoor permutations was given (implicitly) in [MMSK+17]. Here we propose a scheme from any homomorphic one-way function. Examples of homomorphic one-way functions include discrete logarithm and the learning with errors problem [Reg09]. Let $g : \mathcal{D} \to \mathcal{R}$ be a homomorphic one-way function, and let $\mathcal{F}_{\mathsf{anon}}$ be the ideal functionality for an anonymous communication channel. The algorithms of our construction are given in Fig. 5.4. Note that KeyGen simply returns the users' identities, and thus, it is omitted.

In the setup algorithm, the user U_0 initializes the AMHL by sampling n values $(y_0, \ldots, y_{n-1})$ from the domain of g. Then it sends (via $\mathcal{F}_{\mathsf{anon}}$) a triple $(g(\sum_{j=0}^{i-1} y_j), g(\sum_{j=0}^{i} y_j), y_i)$ to each intermediate user. The intermediate user U_i can then check that the triple is well formed using the homomorphic properties of g. Two contiguous users U_i and U_{i+1} can agree on the shared value of $\ell_i := Y_i = g(\sum_{j=0}^{i} y_j)$ by simply comparing the second and first element of their triple, respectively. Note that publishing a valid opening key k such that $g(k) = \ell$ corresponds to inverting the one-way function g. The opening of the locks can be triggered by the last node in the chain U_n: The initial key $k_n := \sum_{i=0}^{n-1} y_i$ consists of a valid pre-image of $\ell_{n-1} := Y_{n-1}$. As soon as the "right" lock is released, each intermediate user U_i has enough information to release

its "left" lock. To see this, observe that $g(k_{i+1} - y_i) = g(\sum_{j=0}^{i} y_i - y_i) = g(\sum_{j=0}^{i-1} y_i) = Y_{i-1}$. For the security of the construction, we state the following theorem. Due to space constraints, the proof is deferred to Appendix D.5.

Theorem 2. *Let g be a homomorphic one-way function, then the construction in Fig. 5.4 UC-realizes the ideal functionality $\mathcal{F}$ in the $(\mathcal{F}_{\mathsf{syn}}, \mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{anon}})$-hybrid model.*

The generic construction presented here requires a cryptocurrency supporting scripts that define (linearly) homomorphic operations. This construction is therefore of special interest in blockchain technologies such as Ethereum [eth] and Hyperledger Fabric [ABB+18], where any user can freely deploy a smart contract without restrictions in the cryptographic operations available. We stress that any function with homomorphic properties is suitable to implement our construction. For instance, lattice-based functions (e.g., from the learning with errors problem) can be used for applications where post-quantum cryptography is required. However, many cryptocurrencies, led by Bitcoin, do not support unrestricted scripts and the deployment of generic AMHLs requires non-trivial changes (i.e., a hard fork). To overcome this challenge, we turn our attention to scriptless AMHLs, where a signature scheme can simultaneously be used for authorization and locking.

5.5.3 Scriptless Schnorr-based Construction

The crux of a scriptless locking mechanism is that the lock can consist only of a message m and a public key pk of a given signature scheme and can be released only with a valid signature σ of m under pk. Scriptless locks stem from an idea of Poelstra [Poea], who proposed a way to embed contracts into Schnorr signatures. In this work, we cast Poelstra's informal idea in our framework, and we formally characterize its security and privacy guarantees. We further optimize this scheme in order to save one round of communication.

Recall that a public key in a Schnorr signature consists of an element $Q := x \cdot G$ and a signature $\sigma := (k \cdot G, s)$ on a message m is generated by sampling $k \leftarrow\!\!\$\, \mathbb{Z}_q$, computing $e := H(Q \| k \cdot G \| m)$, and setting $s := k - xe$. On a very high level, the locking mechanism consists of an "incomplete" distributed signing of some message m: Two users U_i and U_{i+1} agree on a randomly chosen element $R_0 + R_1$ using a coin tossing protocol, then they set the randomness of the signature to be $R := R_0 + R_1 + Y_i$. Next they jointly compute the value $s := r_0 + r_1 + e \cdot (x_0 + x_1)$ as if Y_i was not part of the randomness, where e is the hash of the transcript so far. The resulting (R, s) is *not* a valid signature on m, since the additive term y^* (where $y^* \cdot G = Y_i$) is missing from the computation of s. However, once the discrete logarithm of Y_i is revealed, a valid signature m can be computed by U_{i+1}. Leveraging this observation, we can enforce an *atomic* opening: The subsequent locking (between U_{i+1} and U_{i+2}) is conditioned on some $Y_{i+1} = Y_i + y_{i+1} \cdot G$. This way, the opening of the right lock reveals the value $y^* + y_{i+1}$ and U_{i+1} can immediately extract y^* and open its left lock with a valid signature on m. We defer the formal description and the analysis of the scheme to Appendix D.3.

5.5.4 Scriptless ECDSA-based Construction

The Schnorr-based scheme is limited to cryptocurrencies that use Schnorr signatures to authorize transactions and thus is not compatible with those systems, prominently Bitcoin, that implement ECDSA signatures. Therefore, an ECDSA-based scriptless AMHL is interesting both from a practical and a theoretical perspective as to whether it can be done at all. Prior to our work, the existence of such a construction was regarded as an open question [Poeb]. The core difficulty is that the Schnorr-based construction exploits the linear structure of the signature, whereas the ECDSA signing algorithm completely breaks this linearity feature (e.g., it requires to compute multiplicative shares of a key and the inverse of elements within a group). In the following, we show how to overcome these problems, introducing an ECDSA-based construction for AMHLs: Locks are of the form (pk, m) and can only be opened with an ECDSA signature σ on m under pk.

Let $\mathbb{G}$ be an elliptic curve group of order q with base point G and let $H : \{0,1\}^* \to \{0,1\}^{|q|}$ be a hash function. The ECDSA-based construction is shown in Fig. 5.5. Each pair of users (U_i, U_j) generates a shared ECDSA public key $\mathsf{pk} = (x_i \cdot x_j) \cdot G$ via the $\mathcal{F}_{\mathsf{kgen}}^{\mathsf{ECDSA}}$ functionality. Additionally, U_i receives a Paillier secret key sk and his share x_i, whereas and U_j receives the share x_j and the Paillier encryption c of x_i. The key generation functionality is fully described in Appendix D.5.

The setup here is very similar to the setup of the generic construction in Fig. 5.4 except that the one-way function g is now instantiated with discrete logarithm over elliptic curves. Each intermediate user U_i receives a triple (Y_{i-1}, Y_i, y_i) such that $Y_i := Y_{i-1} + y_i \cdot G$, from $\mathcal{F}_{\mathsf{anon}}$. For technical reasons, the initiator of the AMHL also includes a proof of wellformedness for each Y_i.

The locking algorithm is initiated by two users U_i and U_{i+1} who agree on a message m (which encodes a unique id) and on a value $Y_i := y^* \cdot G$ of unknown discrete logarithm. The two parties then run a coin tossing protocol to agree on a randomness $R = (r_0 \cdot r_1) \cdot Y_i$. When compared to the Schnorr instance, the crucial technical challenge here is that the randomnesses are composed multiplicatively due to the structure of the ECDSA signature and therefore, the trick applied in the Schnorr construction no longer works here. R is computed through a Diffie-Hellman-like protocol, where the parties exchange $r_0 \cdot Y_i$ and $r_1 \cdot Y_i$ and locally recompute R. As before, the shared ECDSA signature is computed by "ignoring" the term Y_i, since the parties are unaware of its discrete logarithm. The corresponding tuple $\left(r_x, s' := \frac{r_x \cdot (x_0 \cdot x_{i+1}) + H(m)}{r_0 \cdot r_1}\right)$ is jointly computed using the encryption of x_0 and the homomorphic properties of Paillier encryption. This effectively means that $(r_x, s') = (r_x, s^* \cdot y^*)$, where (r_x, s^*) is a valid ECDSA signature on m. In order to check the validity of s', the parties additionally need to exchange the value $R^* := (r_0 \cdot r_1) \cdot G = (y^*)^{-1} \cdot R$. The computation of R^* (together with the corresponding consistency proof) is piggybacked in the coin tossing. Given R^*, the validity of s' can be easily verified by both parties by recomputing it "in the exponent".

From the perspective of U_{i+1}, releasing his left lock without a key for his right lock implies solving the discrete logarithm of Y_i. On the converse, once the right lock is released, the value $y^* + y_{i+1}$ is revealed (where y_{i+1} is part of the state of U_{i+1}) and a valid signature can be

computed as $\left(r_x, \frac{s'}{y^*}\right)$. The security of the construction is established by the following theorem (see Appendix D.5 for a full proof).

Theorem 3. *Let* COM *be a secure commitment scheme and let* NIZK *be a non-interactive zero knowledge proof. If ECDSA signatures are strongly existentially unforgeable and Paillier encryption is* ecCPA *secure, then the construction in Fig. 5.5 UC-realizes the ideal functionality* $\mathcal{F}$ *in the* $(\mathcal{F}_{\mathsf{kgen}}^{\mathsf{ECDSA}}, \mathcal{F}_{\mathsf{syn}}, \mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{anon}})$*-hybrid model.*

5.5.5 Hybrid AMHLs

We observe that, when instantiated over the same elliptic curve $\mathbb{G}$, the setup protocols of the Schnorr and ECDSA constructions are identical. This means that the initiator of the lock does not need to know whether each intermediate lock is computed using the ECDSA or Schnorr method. This opens the doors to hybrid AMHLs: Given a unified setup, the intermediate pair of users can generate locks using an arbitrary locking protocol. The resulting AMHL is a chain of (potentially) different locks, and the release algorithm needs to be adjusted accordingly. For the case of ECDSA-Schnorr the user needs to extract the value y^* from the right Schnorr signature (R^*, s^*) and his state $s^R := s' = s^* - y^* + y_{i+1}$ and $s^I := (Y_i, Y_{i+1}, y_{i+1})$. Given y^*, he can factor it out of its left state $s^L = ((r, s \cdot y^*), m, \mathsf{pk})$ and recover a valid ECDSA signature.

The complementary case (Schnorr-ECDSA) is handled mirroring this algorithm. Similar techniques also apply to the generic construction when the one-way function is instantiated appropriately (i.e., with discrete logarithm over the same curve). This flexibility enables atomic swaps and cross-currency payments (see Section 5.7). The security for the hybrid AMHLs follows similar to the standard case.

5.6 Performance Analysis

5.6.1 Implementation Details

We have developed a prototypical Python implementation to demonstrate the feasibility of our construction and evaluate its performance. We have implemented the cryptographic operations required by AMHLs as described in this work. We have used the Charm library [Cha] for the cryptographic operations. We have instantiated ECDSA over the elliptic curve *secp256k1* (the one used in Bitcoin), and we have implemented the homomorphic one-way function as $g(x) := x \cdot G$ over the same curve. Zero-knowledge protocols for discrete logarithms have been implemented using Σ protocols [Dam02] and made non-interactive using the Fiat-Shamir heuristic [FS86]. For a commitment scheme, we have used SHA-256 modeled as a random oracle [BR93].

5.6.2 Evaluation

Testbed. We conducted our experiments on a machine with an Intel Core i7, 3.1 GHz, and 8 GB RAM. We consider the Setup, Lock, Rel and Vf algorithms. We do not consider KGen as we use off-the-shelf algorithms without modification. Moreover, the key generation is executed only once upon creating a link and thus does not affect the online performance of AMHLs. We refer

to [Lin17] for a detailed performance evaluation of the ECDSA key generation. The results of our performance evaluation are shown in Table 5.1.

Computation Time. We measure the computation time required by the users to perform the different algorithms. For the case of two-party protocols (e.g., Setup and Lock), we consider the time for the two users together. We make two main observations: First, the script-based construction based on discrete logarithm is faster than scriptless AMHLs. Second, all the algorithms require computation time of at most 60 milliseconds on commodity hardware.

Communication Overhead. We measure the communication overhead as the amount of information that users need to exchange during the execution of interactive protocols, in particular, Setup and Lock. As expected, the generic construction based on discrete logarithm requires less communication overhead than scriptless constructions. The scriptless construction based on ECDSA requires a higher communication overhead. This is mainly due to having the signing key distributed multiplicatively and a more complex structure of the final signature when compared to the Schnorr approach.

Computation Cost. We measure the computation cost in terms of the gas required by a smart contract implementing the corresponding algorithm in Ethereum. Naturally, we consider this cost only for the generic approach based on discrete logarithm. We observe that setting up the corresponding contract requires 350849 units of gas per hop. At the time of writing, each AMHL, therefore, costs considerably less than 0.01 USD.

Application Overhead. We measure the overhead incurred by the application in terms of the memory required to handle application-dependent data, i.e., information defining the lock and the opening. In tune with the rest of measurements, the generic construction based on discrete

		Generic	**Schnorr**	**ECDSA**
Setup	Time (ms)	$0.3 \cdot n$	$1 \cdot n$	$1 \cdot n$
	Comm (bytes)	$96 \cdot n$	$128 \cdot n$	$128 \cdot n$
Lock	Time (ms)	–	2	60
	Comm (bytes)	32	256	416
Rel	Time (ms)	–	0.002	0.02
	Comm (bytes)	0	0	0
Vf	Time (ms)	–	0.6	0.06
	Comm (bytes)	0	0	0
Comp Cost (gas)		$350849 \cdot n$	0	0
Lock size (bytes)		32	$32 + \|m\|$	$32 + \|m\|$
Open size (bytes)		32	64	64

Table 5.1: Comparison of the resources required to execute the algorithms for the different AMHLs. We denote by n the length of the path. We denote the negligible computation times by – (e.g., single memory read). We denote the size of an application-dependent message by $|m|$ (e.g., a transaction in a payment-channel network).

logarithms requires the smallest amount of memory, both for lock and opening information. The different scriptless approaches require the same amount of memory from the application.

Scalability. We study the running time and communication overhead required by each of the roles in a multi-hop lock protocol (i.e., sender, receiver, and intermediate user). We consider only the generic approach and the ECDSA construction as representative of the scriptless approach. In the absence of significant metrics from current PCNs, we consider a path length of ten hops as suggested for similar payment networks such as the Ripple credit network [MMSKM17].

Regarding the computation time, the sender requires 3ms with the generic approach and 10ms with the ECDSA scriptless approach. The computation time at intermediate users remains below 1ms for ECDSA and negligible with the generic approach as they only have to check the consistency of the locks with the predecessor and the successor, independently of the length of the path. Similarly, the computation overhead of the receiver remains below 1ms as she only checks if a given key is valid to open the lock according to the Vf algorithm. In summary, a non-private payment over a path of 5 users takes over 600ms as reported in [MMSK$^+$17]. Extending it with the constructions presented in this work provides formal privacy guarantees at virtually no overhead.

Regarding the communication overhead, the sender must send a message of about 960 bytes for the generic approach while about 1280 bytes are required instead if ECDSA scriptless locks are used. Since Sphinx, the anonymous communication network used in the LN requires padded messages at each node to ensure anonymity, we foresee that every intermediate user must forward a message of the same size.

Comparing these results with other multi-hop and privacy-preserving PCNs available in the literature, we make the following observations. First, the overhead for the constructions presented in this work are in tune with TeeChain [LNE$^+$18], where the overhead per hop is about 0.4 ms in a setting where cryptographic operations required for the multi-hop locks have been replaced by a trusted execution environment. Second, our constructions significantly reduce the communication and computation overhead required by multi-hop HTLC [MMSK$^+$17]: While a payment using multi-hop HTLC requires approximately 5 seconds and 17MB of communication, our approach requires only a few milliseconds and less than 1MB.

In summary, the evaluation results show that even with an unoptimized implementation, our constructions offer significant improvements on computation and communication overhead and are ready to be deployed in practice.

5.7 Applications

5.7.1 Payment Channel Networks

AMHLs can be generically combined with a blockchain B to construct a fully-fledged PCN. Loosely speaking, the transformation works as follows: In the first round, the sender sets up the locks running the Setup algorithm, then each pair of intermediate users executes the Lock protocol and establishes the following AMHL contract.

AMHL (Alice, Bob, ℓ, x, t):
1. If Bob produces the condition k such that $\mathsf{Vf}(\ell, k) = 1$ before t days, Alice pays Bob x coins.
2. If t days elapse, Alice gets back x coins.

Where ℓ is the output lock and x and t are chosen as specified in Section 5.2. Note that we have to assume that B supports the Vf algorithm and time management in its script language. The rest of the payment is unchanged except that the intermediate users execute the Rel algorithm to extract a valid key k to claim the corresponding payment. In Appendix D.6, we provide the exact description of the algorithms and we prove the following theorem.

Theorem 4 (Informal)**.** *Let* B *a secure blockchain and let* $\mathbb{L}$ *be a secure AMHL, then we can construct a secure PCN (as defined in [MMSK+17]).*

Note that even though we defined the security of AMHLs in the UC framework, the composition of multiple AMHL instances in one protocol as needed for realizing PCNs does not come for free if those instances have a shared state. Formally, such a shared state can arise from the use of a shared KGen algorithm. Consequently, we need to show for the KGen algorithms of the presented constructions that they behave independently over multiple invocations and finally make use of the JUC theorem [CR03] to obtain the composability result.

This shows that AMHLs are the only cryptographic primitive (except for the blockchain) needed to construct PCNs. The only limitation is that the blockchain needs to support the verification of the corresponding contract in their scripting language (see the discussion above). For this reason, the scriptless-construction are preferred for those blockchains where the scripting language does not support the evaluation of a homomorphic one-way function (such as Bitcoin).

Application to the Lightning Network. When applied to the LN, the ECDSA AMHL construction conveys several advantages: First, it eliminates the security issues existing in the current LN due to the use of the HTLC contract. Second, it reduces the transaction size as a single signature is required per transaction. This has the benefit of lowering the communication overhead, the transaction fees, and the blockchain memory requirements for closing a payment channel. In fact, we have received feedback from the LN community indicating the suitability of our ECDSA-based construction. Moreover, results from the implementation and testing done by LN developers are available [Froa, Frob].

The applicability of our proposals are not restricted to the LN or Bitcoin: There exist other PCNs that could similarly take advantage of the scriptless AMHLs presented in this work. For instance, the Raiden Network has been presented as a payment channel network for solving the scalability issue in Ethereum. The adoption of our ECDSA scriptless AMHLs would bring the same benefits to the Raiden Network.

5.7.2 Atomic Swaps

Assume two users U_0 and U_1 holding coins in two different cryptocurrencies and want to exchange them. An *atomic swap* protocol ensures that either the coins are swapped, or the balances are

untouched, i.e., the exchange must be performed atomically. The widely used protocol for atomic swaps described in [BH] leverages the HTLC contract to perform the swap. In a nutshell, an atomic swap can be seen as a multi-hop payment over a path of the form (U_0, U_1, U_0). This approach inherits the security concerns of the HTLC contract. Scriptless AMHLs also enhance this application domain with formally proven security guarantees.

Additionally, our constructions contribute to the *fungibility* of the coins, a crucial aspect for any available (crypto)currency. Current protocols rely on transactions that are clearly distinguishable from regular payments (i.e., one-to-one payments). In particular, atomic swap transactions contain the HTLC contract, in contrast to regular transactions. Scriptless AMHLs eliminate this issue since even atomic swap transactions only require a single signature from a public key, making them indistinguishable from regular payments. Similar arguments also apply for multi-hop payments in PCNs.

5.7.3 Interoperable PCNs

Among the cryptocurrencies existing today, an interesting problem consists in performing a multi-hop payment where each link represents a payment channel defined in a different cryptocurrency. In this manner, a user with a payment channel funded in a given cryptocurrency can use it to pay to another user with a payment channel in a different cryptocurrency. Currently, the InterLedger protocol [ST] tackles this problem with a mechanism to perform cross-currency multi-hop payments that rely on the HTLC contract, aiming to ensure the payment atomicity across different hops.

However, apart from the already discussed issues associated with HTLC, the InterLedger protocol mandates that all cryptocurrencies implement HTLC contracts. This obviously hinders the deployment of this approach. Instead, it is possible to use the different AMHL constructions presented in this work on a single path, as described in Section 5.5.5, therefore expanding the domain of cross-currency multi-hop payments.

5.8 Related Work

A recent work [DKLas18] shows a protocol to compute an ECDSA signature using multi-party computation. However, it is not as efficient as Lindell's approach [Lin17].

There exists extensive literature proposing constructions for payment channels [LEPS16, DW15, PD, DRO]. These works focus on a single payment channel and their extension to PCNs remain an open challenge. TumbleBit [HAB+17] and Bolt [GM17] support off-chain payments while achieving payment anonymity guarantees. However, the anonymity guarantees of these approaches are restricted to single-hop payments and their extension to support multi-hop payments remains an open challenge.

State channels [DEFM17, MBKM19, KG17] and state channel networks [DFH18] cannot work with prominent cryptocurrencies except Ethereum. TeeChain [LNE+18] requires the availability of a trusted execution environment for each user. Instead, our proposal can be seamlessly deployed

today in virtually all cryptocurrencies, including Ethereum. In addition, AMHL enables operations between different blockchains, which is clearly not the case for Ethereum-only solutions. If we focus on the specific setting of payment channels in Ethereum, AMHL is more efficient (i.e., it requires less gas and bytes) as payment conditions are encoded in the signature and not in additional scripts. Finally, [DEFM17, DFH18] provide a different privacy notion: two endpoints can communicate privately, but the intermediate nodes know that a virtual channel is opened between them. This information is instead concealed with AMHL. Formalizing this privacy leakage and comparing it with our privacy definition is an interesting future work.

The LN has emerged as the most promising approach for PCN in practice. Its current description [ln-b] is being followed by several implementations [lnd, lig18, ecl]. However, these implementations suffer from the security and privacy issues with PCNs as described in this work. Instead, we provide several constructions for AMHLs that can be leveraged to have secure and anonymous multi-hop payments.

Malavolta et al. [MMSK+17] propose a protocol for secure and anonymous multi-hop payments compatible with the current LN. Their approach, however, imposes an overhead of around 5 MB for the nodes in the network, therefore hindering its deployability. Here, we propose several efficient constructions that require only a few bytes of communication.

In the recent literature, we can find proposals for secure and privacy-preserving atomic swaps. Tesseract [BJZ+17] leverages trusted hardware to perform real-time cryptocurrency exchanges. The Merkleized Abstract Syntax Trees (MAST) protocol has been proposed as a privacy solution for atomic swaps [Lau]. However, MAST relies on scripts that are not available in the major cryptocurrencies today. Moreover, specific contracts for atomic swaps hinder the fungibility of the currency: An observer can easily differentiate between a regular payment and a payment resulting from an atomic swap.

5.9 Conclusion

We rigorously study the cryptographic core functionality for security, privacy, and interoperability guarantees in PCNs, presenting a new attack on today's PCNs (the wormhole attack) and proposing a novel cryptographic construction (AMHLs). We instantiate AMHLs in two settings: script-based and scriptless. In the script-based setting, we demonstrate that AMHLs can be realized from any (linear) homomorphic operation. In the scriptless setting, we propose a construction based on ECDSA, thereby catering to the vast majority of cryptocurrencies deployed today. Our performance evaluation shows that AMHLs are practical: All operations take less than 100 milliseconds to run and introduce a communication overhead of fewer than 500 bytes.

We show that AMHLs can be combined in a single path and are of interest in several applications apart from PCNs, such as atomic swaps and interoperable PCNs. In fact, LN developers have implemented and tested AMHL for LN. In the future, we plan to devise cryptographic instantiations of PCNs for the few cryptocurrencies not yet covered, most notably Monero.

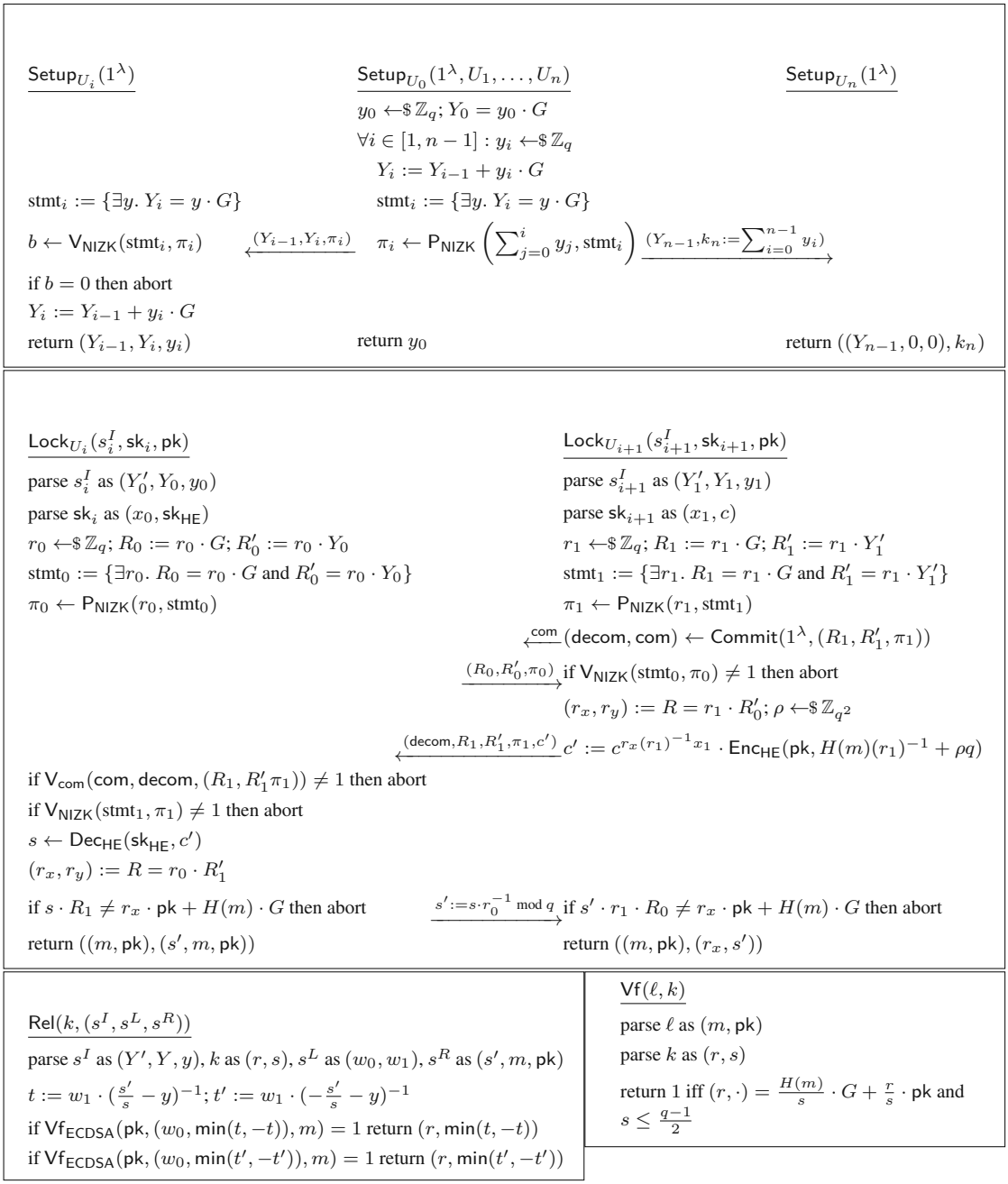

Figure 5.5: Algorithms and protocols for the ECDSA-based construction.

CHAPTER 6

Conclusion and Directions for Future Research

6.1 Conclusion

In this thesis, we have shown how principled theoretical foundations can enhance distributed blockchain applications' security. To this end, we focused on the security analysis of Ethereum smart contracts and off-chain protocols.

First, we presented a formal small-step semantics for Ethereum Virtual Machine bytecode which we also implemented in the proof assistant F*. Using this small-step semantics, we formulated generic security notions for Ethereum smart contracts, most prominently the *call integrity* property for which we presented a designated proof strategy.

Next, we systematized the existing approaches to the analysis of Ethereum smart contracts, focusing on automatic static tools that come with soundness guarantees. We highlighted the specific challenges that emerged in the design and implementation of automated static analyzers for Ethereum smart contracts and illustrated these challenges' severeness by unveiling soundness issues in existing automated static analysis tools that claim soundness.

We then presented the sound static analysis tool *eThor* that overcomes the presented challenges by following a principled design and implementation approach. *eThor* relies on an abstract semantics based on Constrained Horn clauses which devises several advanced domain-specific abstractions and is proven sound against the previously introduced small-step semantics. For a modular design and a reliable generation of a practical tool from the analysis specification, we introduced the specification language *HoRSt* that automatically translates into the generic `smt-lib` format for SMT-solvers. We conducted an empirical evaluation that not only confirms *eThor*'s soundness but also shows that *eThor* outperforms the state-of-the-art analyzer ZEUS (in terms of F-measure) on a real-world data set.

Finally, we showed how to improve the security and privacy in payment channel networks for Bitcoin and other cryptocurrencies. To this end, we first presented a novel attack on the security of Bitcoin's Lightning Network. Next, developed a cryptographic primitive that allows for building secure and anonymous payment channel networks in different cryptocurrencies. In particular, we proved this primitive to be realizable by a construction based on the ECDSA signature scheme, which Bitcoin uses. Further, we showed that different instances of the primitive securely compose to secure multi-hop payments involving participants from other cryptocurrencies.

6.2 Application to Recent Trends in Decentralized Finance

Recently, Decentralized Finance (short DeFi) gained significant importance as an application area of cryptocurrencies. Broadly, DeFi refers to all kinds of financial applications that are realized using blockchain technologies and hence do not rely on a central authority. Canonical examples are exchanges or banking services. The complexity of DeFi applications usually requires the power of an expressive scripting language such as provided by Ethereum. This requirement makes Ethereum the most important DeFi platform. As a consequence, the rise of DeFi came with an increased transaction load on Ethereum [eth21b].

DeFi amplifies the prevalent issues of cryptocurrencies which are discussed in this thesis:

1. DeFi applications are implemented in the form of (interacting) smart contracts, which are particularly security-critical. Bugs in these smart contracts can cause severe financial losses and undermine the promise of DeFi to establish a fair and secure financial market that is accessible for everyone. Known issues, such as the transaction order dependency vulnerabilities (discussed in Section 2.4), gain new significance in the DeFi setting. E.g., one highly debated attack class is the so-called *front running* [Dil21] where attackers take advantage of (already submitted) transactions of another user to preempt that user in their action. But also involved reentrancy attacks, such as the one on UniSwap [Cim20], Ethereum's largest decentralized financial exchange application [Kha20], reoccur, making the verification of smart contracts a predominant topic.

   The results of this thesis present a starting point for the development of powerful, provably sound analysis tools for DeFi applications that come with a high degree of automation. The existing attack landscape shows both the verification of generic properties (such as single-entrancy or independence of miner-controlled parameters), as well as the verification of contract-specific properties (as described in Section 4.2.5) to be relevant. In particular, for complex applications that involve multiple services (as many DeFi applications do), it becomes essential to investigate the security of applications in the interplay with a complex environment of other applications. This setting motivates the bulk verification of several contracts (mentioned as possible extension in Section 4.6), but also the exploration of compositional reasoning techniques for smart contracts. A starting point for the development of such compositional reasoning techniques is the contract invariant checking as enabled by the functional correctness extension of *eThor* presented in Section 4.4.2. One could think of developing high-level techniques (to argue about complex inter-contract properties) to

be finally broken down to invariance checks on individual smart contracts automatically discharged by *eThor*. This vision shows that *eThor* can be an important building block in a future analysis pipeline for smart contracts. Future improvements of *eThor* as will be discussed in Section 6.3.1 would benefit such development.

2. Due to the higher transaction load, scaling solutions such as PCNs discussed in Chapter 5 gain importance. For the DeFi use case, the focus of these solutions lies in the scaling of transactions that invoke intricate smart contract executions. This goal stands in contrast with the case of Bitcoin and many other cryptocurrencies, where (simple) financial transactions between two parties represent a more common use case. For the latter, PCNs offering fast financial peer-to-peer transactions constitute an adequate means of scaling. For scaling contract executions in Ethereum, instead, the concept of *state channel networks* [DFH18] was introduced. This concept, however, comes with the major drawback that (as in the PCN case) two (or multiple) interacting parties need to agree on a setup upfront for (jointly) executing a contract off-chain. This setting countermines the function of many DeFi applications, which allow arbitrary users to interact with the application over time. E.g., a decentralized exchange such as UniSwap is open for any user to register and exchange their funds.

   For this reason, alternative scaling solutions for the case of Ethereum are investigated, most prominently a concept called *rollups* [eth21a]. The idea of rollups is that (given a suitably programmed smart contract) multiple transactions for that contract can be bundled (off-chain) by certain entities (so-called relayers) that then only publish the overall result of these transactions. The relayers to ensure the correctness of state updates either give a zero-knowledge proof or provide enough data so that the update can be challenged and enforced on-chain (in which case the relayer is charged a previously given security deposit). The challenges for rollups differ substantially from those in PCNs. The (secure) implementation of rollups relies on a smart contract applying the rolled-up changes and implementing the security checks (either the check of zero-knowledge proofs or the challenge mechanism). This use of a smart contract as the central enforcement mechanism moves the complexity of rollups (compared to PCNs) from the cryptographic protocol layer to the design and implementation of the contract. Critical questions are how to realize the full expressivity of the EVM in the setting of rollups, how to minimize published data, and how relaying transactions may open up room for new (mechanistic) attacks.

   Even though these are fascinating and relevant research questions, and rollups offer promising solutions to shortcomings of PCNs (such as the need for locking collaterals, or the requirement of users to stay online) the purpose of rollups is partly orthogonal to the one of the PCNs discussed in Chapter 5. While rollups both *enable* and *require* computations in an expressive language, the focus of Chapter 5 is a framework for secure off-chain payments in settings with *restricted scripting languages*. Even though coming with the priorly discussed limitations of payment channels, the proposed design offers a small common denominator for an off-chain scaling solution in different existing cryptocurrencies. Hence, this solution enables interoperable scaling across multiple currencies for one restricted functionality (payments). It is an exciting challenge to add additional functionality to such a framework

without losing interoperability (so by putting minimal requirements on the underlying scripting language). We aim to investigate this question in future research, as detailed out in Section 6.3.4.

6.3 Directions for Future Work

With the automated static analysis tool *eThor* presented in Chapter 4 we do not only offer a solution to the verification of specific security properties of Ethereum smart contracts. We also introduce a generic and extensible framework for the design of static analyzers that facilitates the reasoning about the resulting tools' theoretical guarantees.

This framework calls for extensions along several dimensions:

6.3.1 Extensions of *eThor*

We plan to enlarge the scope of *eThor* itself by refining the abstractions to allow for more precise analysis and by encoding other generic security properties.

One possible refinement of the analysis could be a domain-specific storage abstraction that accounts for the hash-based allocation scheme used by the *Solidity* compiler to represent complex data types in the persistent storage. This refinement would allow for a more precise analysis of smart contracts compiled from *Solidity* and using maps or arrays. Further, we could improve the analysis's precision by adding support for the bulk-verification of several smart contracts to more precisely analyze smart contracts that use (trusted) library contracts.

For expressing additional security properties within *eThor*– such as the atomicity property in Section 2.4 – it is currently required to soundly approximate these notions as reachability properties. This approximation, however, is still an interesting open research question since most of these properties are hyperproperties that are hard to abstract in terms of reachability.

6.3.2 Extensions of *HoRSt*

In addition to extending the *eThor* tool, we want to expand the *HoRSt* framework to include new features that allow for the generation of more expressive, more performant, and more reliable analysis tools.

Additional Properties An alternative to finding reachability abstractions for relational properties is the extension of the analysis language *HoRSt* to a broader set of properties. An interesting extension would be two-safety properties that would enable the reasoning about two execution traces' relation. One could realize this extension by an automatic generation of cross-product constructions on the implemented abstract semantic rules. Such an extension would come with the advantage that the transformation's correctness could be covered by a universal proof and would not charge further proof obligations to the analysis designer.

New Backends To improve the overall performance of the generated analysis tools, we want to extend *HoRSt* with other translation targets. Not all program classes require the full power of SMT-solvers. As illustrated by the example of the control-flow reconstruction in Section 4.4, some problems fall into more manageable categories and can be way more efficiently solved using other solving strategies. For this reason, we want to specify translations from (well-defined fragments of) *HoRSt* into different output formats such as *Soufflé*. This approach would allow users to conveniently combine the powers of different solvers on their static analysis tasks.

Support for Mechanized Proofs To strengthen the theoretical guarantees of analysis tools based on a *HoRSt* specification, we plan to extend *HoRSt* with an export functionality for proof assistants such as F*. This extension would enable analysis designers to conduct machine-checked soundness proofs of the analysis, and as a consequence, strengthen the theoretical guarantees of the analysis tools. Soundness proofs for the analysis of real-world languages are lengthy, tedious, and hence error-prone when done on paper. Simultaneously, such proofs often share a similar structure and contain an abundance of redundant cases. For this reason, using proof assistants with high support for automation like F* has the potential to automate large parts of such proofs, in particular when carefully modeling and providing suitable proof infrastructure. A direct export functionality would further come with the advantage that the same source would generate both the tool and the formalization that serves as a basis for the proof. This would create a close connection between the proof and the tool.

6.3.3 New Analysis Targets

The *HoRSt* framework is not bound to the use case of EVM bytecode analysis but can serve as a basis for static analyzers for other languages. Consequently, all such tools would benefit from the general improvements of the framework discussed in the last paragraphs. Immediate application domains would be the new Ethereum smart contract language eWASM or smart contracts for the rising EOSIO blockchain. Above that, the applicability of the *HoRSt* framework is by far not limited to the blockchain context. It could also help develop reliable analysis tools for languages from other domains such as WebAssembly, PHP, or RUST.

In addition to the presented research opportunities in the field of automated static analysis, another exciting research direction would be to bridge the two different forms of distributed blockchain applications discussed in this thesis. Building on the insights on smart contracts and off-chain protocols, I plan to study *off-chain smart contracts* – off-chain protocols whose functionality goes beyond payments.

6.3.4 Off-chain Smart Contracts

Recent works [DFH18] demonstrate how to develop off-chain smart contracts when relying on expressive scripting languages (such as the one of Ethereum). However, it is still an open problem to characterize the off-chain functionality achievable when building on more basic scripting languages (such as supported by Bitcoin). Also, [BZ18, BMZ20] showed how to extend the expressiveness of blockchain systems with limited scripting languages by adding an (on-chain)

protocol layer. These two results indicate that the expressiveness of blockchain scripting languages should be considered in several dimensions: Their computational expressiveness, the class of protocols that are expressible with smart contracts, as (well-behaved) protocol participants, as well as the form of off-chain protocols that can be built from these languages. For this reason, it would be interesting to explore which off-chain functionality can be realized depending on the underlying scripting language and what could be suitable candidates for languages to capture such off-chain functionality. The design of such off-chain languages and provably sound verification tools would be an exciting line of research requiring the combination of static verification techniques and cryptographical reasoning on the protocol level.

List of Figures

List of Tables

Bibliography

[ABB+18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *EuroSys*, pages 30:1–30:15, 2018.

[ABBS18] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. *CPP. ACM. To appear*, 2018.

[ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*, pages 164–186. Springer, 2017.

[ACG+19] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. Safevm: a safety verifier for ethereum smart contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 386–389, 2019.

[Adh17] Chandra Adhikari. Secure framework for healthcare data management using ethereum-based blockchain technology. 2017.

[AEP18] Shaun Azzopardi, Joshua Ellul, and Gordon J Pace. Monitoring smart contracts: Contractlarva and open challenges beyond. In *International Conference on Runtime Verification*, pages 113–137. Springer, 2018.

[AEVL16] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. Medrec: Using blockchain for medical data access and permission management. In *Open and Big Data (OBD), International Conference on*, pages 25–30. IEEE, 2016.

[ALV] MARIO M ALVAREZ. Elle: Foundationally verified compilation for ethereum.

[AMJC20] Imran Ashraf, Xiaoxue Ma, Bo Jiang, and WK Chan. Gasfuzzer: Fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities. *IEEE Access*, 2020.

[Ari19] Emilio Jesús Gallego Arias. Towards principled compilation of ethereum smart contracts (sok). In *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2019.

[bam] Bamboo. Available at `https://github.com/pirapira/bamboo`.

[BB17] Alex Beregszaszi and Paweł Bylica. Eip-145: Bitwise shifting instructions in evm. `https://eips.ethereum.org/EIPS/eip-145`, February 2017. Ethereum Improvement Proposals, no. 145 [Online serial].

[BD77] Rod M Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1):44–67, 1977.

[BDLF+16a] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96. ACM, 2016.

[BDLF+16b] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96. ACM, 2016.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Symposium on Theory of Computing*, pages 103–112, 1988.

[BGL+20] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *PLDI*, pages 454–469, 2020.

[BGM16] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies without proof of work. In *International conference on financial cryptography and data security*, pages 142–157. Springer, 2016.

[BGM19] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. A minimal core calculus for solidity contracts. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 233–243. Springer, 2019.

[BGMR15] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, pages 24–51. Springer, 2015.

[BH] Sean Bowe and Daira Hopwood. Hashed time-locked contract transactions. Bitcoin Improvement Proposal. https://github.com/bitcoin/bips/blob/master/bip-0199.mediawiki.

[bit] Blockchain explorer information. https://blockchain.info/.

[BJZ+17] Iddo Bentov, Yan Ji, Fan Zhang, Yunqi Li, Xueyuan Zhao, Lorenz Breidenbach, Philip Daian, and Ari Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In *ePrint Archive*, page 1153, 2017.

[BKM+13] M. Backes, A. Kate, P. Manoharan, S. Meiser, and E. Mohammadi. Anoa: A framework for analyzing anonymous communication protocols. In *CSF*, pages 163–178, 2013.

[BKT17] Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. Findel: Secure derivative contracts for ethereum. In *International Conference on Financial Cryptography and Data Security*, pages 453–467. Springer, 2017.

[BM17] Alex Beregszaszi and Nikolai Mushegian. Eip-140: Revert instruction. https://eips.ethereum.org/EIPS/eip-140, February 2017. Ethereum Improvement Proposals, no. 140 [Online serial].

[BMZ20] Massimo Bartoletti, Maurizio Murgia, and Roberto Zunino. Renegotiation and recursion in bitcoin contracts. In *International Conference on Coordination Languages and Models*, pages 261–278. Springer, 2020.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS*, 1993.

[BR17] Vitalik Buterin and Christian Reitwiessner. Eip-214: New opcode staticcall. https://eips.ethereum.org/EIPS/eip-214, February 2017. Ethereum Improvement Proposals, no. 214 [Online serial].

[But16] Vitalik Buterin. Eip-170: Contract code size limit. https://eips.ethereum.org/EIPS/eip-170, November 2016. Ethereum Improvement Proposals, no. 170 [Online serial].

[But18] Vitalik Buterin. Eip-1014: Skinny create2. https://eips.ethereum.org/EIPS/eip-1014, April 2018. Ethereum Improvement Proposals, no. 1014 [Online serial].

[BZ18] Massimo Bartoletti and Roberto Zunino. Bitml: a calculus for bitcoin smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 83–100, 2018.

[Can01] R Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–, 2001.

[CC04] Patrick Cousot and Radhia Cousot. Basic concepts of abstract interpretation. In *Building the Information Society*, pages 359–366. Springer, 2004.

[CDE+16] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On Scaling Decentralized Blockchains. In *FC*, pages 106–125, 2016.

[CDPZ19] Silvia Crafa, Matteo Di Pirro, and Elena Zucca. Is solidity solid enough? In *International Conference on Financial Cryptography and Data Security*, pages 138–153. Springer, 2019.

[cfg20] evm-cfg-builder. https://github.com/crytic/evm_cfg_builder, 2020.

[Cha] Charm: A framework for rapidly prototyping cryptosystems. `https://github.com/JHUISI/charm`.

[Cim20] Catalin Cimpanu. Hackers steal $25 million worth of cryptocurrency from lendf.me platform. `https://www.zdnet.com/article/hackers-steal-25-million-worth-of-cryptocurrency-from-uniswap-and-lendf-me/`, 2020.

[CKY18] Jason Paul Cruz, Yuichi Kaji, and Naoto Yanai. Rbac-sc: Role-based access control using smart contract. *Ieee Access*, 6:12240–12251, 2018.

[CL05] Jan Camenisch and Anna Lysyanskaya. A formal treatment of onion routing. In *CRYPTO*, 2005.

[CLL] Thomas Cook, Alex Latham, and Jae Hyung Lee. Dappguard: Active monitoring and defense for solidity smart contracts.

[Cob17] Michael Coblenz. Obsidian: A safer blockchain programming language. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pages 97–99. IEEE, 2017.

[CR03] Ran Canetti and Tal Rabin. Universal composition with joint state. In *Annual International Cryptology Conference*, pages 265–281, 2003.

[Dam02] Ivan Damgård. On σ-protocols. *Lecture Notes, University of Aarhus, Department for Computer Science*, 2002.

[DAS19] Monika Di Angelo and Gernot Salzer. A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 69–78. IEEE, 2019.

[DEFM17] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment hubs over cryptocurrencies. In *ePrint*, 2017.

[DFH18] Stefan Dziembowski, Sebastian Faust, and Kristina Hostakova. General state channel networks. In *CCS*, 2018.

[Dil21] ConsenSys Diligence. Known attacks - front-running. https://consensys.github.io/smart-contract-best-practices/known_attacks/#front-running, 2021.

[DKLas18] J. Doerner, Y. Kondi, E. Lee, and a. shelat. Secure two-party threshold ecdsa from ecdsa assumptions. In *S&P*, 2018.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DRO] Christian Decker, Rusty Russel, and Olaoluwa Osuntokun. eltoo: A simple layer2 protocol for bitcoin. https://blockstream.com/eltoo.pdf.

[DW15] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Stabilization, Safety, and Security of Distributed Systems*, 2015.

[DWA+17] Changyu Dong, Yilei Wang, Amjad Aldweesh, Patrick McCorry, and Aad van Moorsel. Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. 2017.

[ecl] A scala implementation of the lightning network. https://github.com/ACINQ/eclair.

[Ell20] Joshua Ellul. Towards configurable and efficient runtime verification of blockchain based smart contracts at the virtual machine level. In *International Symposium on Leveraging Applications of Formal Methods*, pages 131–145. Springer, 2020.

[eth] Ethereum website. https://www.ethereum.org/.

[Eth18] Ethereum. *Solidity Docs*, 2018.

[Eth19] Ethereum. *Solidity Docs*, 2019.

[eth21a] ethereum.org. Layer 2 scaling - rollups. https://ethereum.org/en/developers/docs/layer-2-scaling/#rollups, 2021.

[eth21b] etherscan.io. Ethereum daily transactions chart. https://etherscan.io/chart/tx, 2021.

[evm] Consensus test suite. Available at https://github.com/ethereum/tests.

[ext20] eThor: extended version, source code, build, and evaluation artifacts. https://secpriv.wien/ethor, 2020.

[FAH20] Joel Frank, Cornelius Aschermann, and Thorsten Holz. {ETHBMC}: A bounded model checker for smart contracts. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[FGG19] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, 2019.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3—where programs meet provers. In *European Symposium on Programming*, pages 125–128. Springer, 2013.

[Froa] Conner Fromknecht. Instantiating scriptless 2p-ecdsa: fungible 2-of-2 multisigs for bitcoin today. Talk at ScalingBitcoin 2018. `https://tokyo2018.scalingbitcoin.org/transcript/tokyo2018/scriptless-ecdsa`.

[Frob] Conner Fromknecht. tpec: 2p-ecdsa signatures. Github repository. `https://github.com/cfromknecht/tpec`.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the Theory and Application of Cryptographic Techniques*, 1986.

[fst] F*. Available at `https://fstar-lang.org`.

[GAGG+17] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts. *Proceedings of the ACM on Programming Languages*, 2(POPL):48, 2017.

[GJKR07] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20(1):51–83, 2007.

[GM17] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In *CCS*, 2017.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[GMS18a] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. Foundations and tools for the static analysis of ethereum smart contracts. In *Proceedings of the 30th International Conference on Computer-Aided Verification (CAV)*, pages 51–78. Springer, 2018.

[GMS18b] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *Proceedings of the 7th International Conference on Principles of Security and Trust (POST)*, pages 243–269. Springer, 2018.

[GMS18c] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts - technical report, 2018. Available at `https://secpriv.tuwien.ac.at/tools/ethsemantics`.

[Gri21] Ilya Grishchenko. Static analysis of low-level code, 2021.

[GY18] Hisham S Galal and Amr M Youssef. Verifiable sealed-bid auction on the ethereum blockchain. In *International Conference on Financial Cryptography and Data Security*, pages 265–278. Springer, 2018.

[HAB+17] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. TumbleBit: An untrusted bitcoin-compatible anonymous payment hub. In *NDSS*, 2017.

[har16] Hard fork after the DAO hack, 2016. Available at `https://blog.ethereum.org/2016/07/26/onward_from_the_hard_fork/`.

[HB12] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 157–171. Springer, 2012.

[HBC+12] Manuel V Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F Morales, and Germán Puebla. An overview of ciao and its design philosophy. *Theory and Practice of Logic Programming*, 12(1-2):219–252, 2012.

[HBDM11] Kryštof Hoder, Nikolaj Bjørner, and Leonardo De Moura. μz–an efficient engine for fixed points with constraints. In *International Conference on Computer Aided Verification*, pages 457–462. Springer, 2011.

[Hir17a] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *1st Workshop on Trusted Smart Contracts*, 2017.

[Hir17b] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*, pages 520–535. Springer, 2017.

[HS09] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal on Information Security*, 8(6):399–422, 2009.

[HSLC17] Adam Hahn, Rajveer Singh, Chen-Ching Liu, and Sijie Chen. Smart contract-based campus demonstration of decentralized transactive energy auctions. In *2017 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT)*, pages 1–5. IEEE, 2017.

[HSR+18] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. pages 204–217. IEEE, July 2018.

[JB18] Nick Johnson and Paweł Bylica. Eip-1052: Extcodehash opcode. `https://eips.ethereum.org/EIPS/eip-1052`, May 2018. Ethereum Improvement Proposals, no. 1052 [Online serial].

[JKL+18] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. Executable operational semantics of solidity. *arXiv preprint arXiv:1804.01295*, 2018.

[JSS16] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, pages 422–430. Springer, 2016.

[KG17] Rami Khalil and Arthur Gervais. Revive: Rebalancing off-blockchain payment networks. In *CCS*, pages 439–453, 2017.

[KGDS18a] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. NDSS, 2018.

[KGDS18b] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: Analyzing Safety of Smart Contracts. Internet Society, 2018.

[KGM+19] Theodoros Kasampalis, Dwight Guth, Brandon Moore, Traian Florin Șerbănuță, Yi Zhang, Daniele Filaretti, Virgil Șerbănuță, Ralph Johnson, and Grigore Roşu. Iele: A rigorously designed language and tool ecosystem for the blockchain. In *International Symposium on Formal Methods*, pages 593–610. Springer, 2019.

[Kha20] Olga Kharif. Defi boom makes uniswap most sought-after crypto exchange. `https://www.bloomberg.com/news/articles/2020-10-16/defi-boom-makes-uniswap-most-sought-after-crypto-exchange`, 2020.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In *Theory of cryptography*, pages 477–498. 2013.

[Lau] Johnson Lau. Merkelized abstract syntax tree. Bitcoin Improvement Proposal. `https://tinyurl.com/yc9jh6lv`.

[LCO+] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. An analysis tool for smart contracts. Available at https://github.com/melonproject/oyente.

[LCO+16a] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[LCO+16b] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[LEPS16] Joshua Lind, Ittay Eyal, Peter R. Pietzuch, and Emin Gün Sirer. Teechan: Payment channels using trusted execution environments. 2016. http://arxiv.org/abs/1612.07766.

[Les99] Lawrence Lessig. Code is law. *The Industry Standard*, 18, 1999.

[LGTS] SIFIS LAGOUVARDOS, NEVILLE GRECH, ILIAS TSATIRIS, and YANNIS SMARAGDAKIS. Precise static modeling of ethereum "memory".

[lig18] c-lightning – a lightning network implementation in c. Acceses in May 2018. https://github.com/ElementsProject/lightning.

[Lin17] Yehuda Lindell. Fast Secure Two-Party ECDSA Signing. In *CRYPTO*, pages 613–644, 2017.

[LL19] Jing Liu and Zhentian Liu. A survey on security verification of blockchain smart contracts. *IEEE Access*, 7:77894–77904, 2019.

[LLC+18] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68. IEEE, 2018.

[ln-a] 5 potential use cases for bitcoin's lightning network. https://tinyurl.com/y6u4tnda.

[ln-b] Lightning network specifications. https://github.com/lightningnetwork/lightning-rfc.

[lnd] Lightning network daemon. https://github.com/lightningnetwork/lnd.

[LNE+18] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Peter R. Pietzuch, and Emin Gün Sirer. Teechain: Reducing storage costs on the blockchain with offline payment channels. In *Systems and Storage Conference*, page 125, 2018.

[LNZ+16] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *CCS*, pages 17–30, 2016.

[LWX+19] Zixin Li, Haoran Wu, Jiehui Xu, Xingya Wang, Lingming Zhang, and Zhenyu Chen. Musc: A tool for mutation testing of ethereum smart contract. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1198–1201. IEEE, 2019.

[LWZ+19] Ning Lu, Bin Wang, Yongxin Zhang, Wenbo Shi, and Christian Esposito. Neucheck: A more practical ethereum smart contract security analysis tool. *Software: Practice and Experience*, 2019.

[MBKM19] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment channels that go faster than lightning. In *FC*, 2019.

[MFSH17] Patrick McCorry, Siamak F. Shahandashti, and Feng Hao. A smart contract for boardroom voting with maximum voter privacy. *Proceedings of the Financial Cryptography and Data Security Conference*, 2017.

[ML18] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In *International Conference on Financial Cryptography and Data Security*, pages 523–540. Springer, 2018.

[MM17] Florian Mathieu and Ryno Mathee. Blocktix: Decentralized event hosting and ticket distribution network. 2017. Available at `https://blocktix.io/public/doc/blocktix-wp-draft.pdf`.

[MMS+18] Pedro Moreno-Sanchez, Navin Modi, Raghuvir Songhela, Aniket Kate, and Sonia Fahmy. Mind your credit: Assessing the health of the ripple credit network. In *WWW*, pages 329–338, 2018.

[MMSH16] Patrick McCorry, Malte Möser, Siamak Fayyaz Shahandashti, and Feng Hao. Towards bitcoin payment networks. In *ACISP*, 2016.

[MMSK+17] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In *CCS*, 2017.

[MMSKM17] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. SilentWhispers: Enforcing security and privacy in credit networks. In *NDSS*, 2017.

[MMSS+19] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous multi-hop locks for blockchain scalability and interoperability. In *NDSS*, 2019.

[MOA+20] Matteo Marescotti, Rodrigo Otoni, Leonardo Alt, Patrick Eugster, Antti EJ Hyvärinen, and Natasha Sharygina. Accurate smart contract verification through direct modelling. In *International Symposium on Leveraging Applications of Formal Methods*, pages 178–194. Springer, 2020.

[MSH17] Patrick McCorry, Siamak F Shahandashti, and Feng Hao. A smart contract for boardroom voting with maximum voter privacy. In *International Conference on Financial Cryptography and Data Security*, pages 357–375. Springer, 2017.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. Available at `http://bitcoin.org/bitcoin.pdf`.

[NGW17] Benedikt Notheisen, Magnus Gödde, and Christof Weinhardt. Trading stocks on blocks-engineering decentralized markets. In *International Conference on Design Science Research in Information Systems*, pages 474–478. Springer, 2017.

[NKS+18] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. *arXiv preprint arXiv:1802.06038*, 2018.

[oB18] Trail of Bits. Manticore: Symbolic execution for humans. 2018.

[O'C17] Russell O'Connor. Simplicity: A new language for blockchains. *arXiv preprint arXiv:1711.03028*, 2017.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238, 1999.

[par17a] The parity wallet breach, 2017. Available at `https://www.coindesk.com/30-million-ether-reported-stolen-parity-wallet-breach/`.

[par17b] The parity wallet vulnerability, 2017. Available at `https://paritytech.io/blog/security-alert.html`.

[PD] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. Technical Report. `https://lightning.network/lightning-network-paper.pdf`.

[PE] Jack Pettersson and Robert Edström. Safer smart contracts through type-driven development.

[PH10] Andreas Pfitzmann and Marit Hansen. A Terminology for Talking about Privacy by Data Minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management. https://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf, August 2010. v0.34.

[PM18] Adrian-Tudor Panescu and Vasile Manta. Smart contracts for research data rights management over the ethereum blockchain network. *Science & Technology Libraries*, 37(3):235–245, 2018.

[Poea] Andrew Poelstra. Lightning in scriptless scripts. Mailing list post. `https://lists.launchpad.net/mimblewimble/msg00086.html`.

[Poeb] Andrew Poelstra. Scriptless scripts. Presentation slides. `https://download.wpsoftware.net/bitcoin/wizardry/mw-slides/2017-05-milan-meetup/slides.pdf`.

[PZS+18] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. A formal verification tool for ethereum vm bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 912–915. ACM, 2018.

[QHC+18] Meixun Qu, Xin Huang, Xu Chen, Yi Wang, Xiaofeng Ma, and Dawei Liu. Formal verification of smart contracts from the perspective of concurrency. In *International Conference on Smart Blockchain*, pages 32–43. Springer, 2018.

[rai] Raiden network. `http://raiden.network/`.

[Rc10] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):34, 2009.

[Rei17] Christian Reitwiessner. Eip-211: New opcodes: Returndatasize and returndatacopy. `https://eips.ethereum.org/EIPS/eip-211`, February 2017. Ethereum Improvement Proposals, no. 211 [Online serial].

[RMKG18] Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. Settling payments fast and private: Efficient decentralized routing for path-based transactions. In *NDSS*, 2018.

[saf19] SafeMath library source. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol, 2019.

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

[SED] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. Writing safe smart contracts in flint.

[SGG+14] Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. Checking probabilistic noninterference using joana. *it - Information Technology*, 56:280–287, November 2014.

[SGSM20] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. ethor: Practical and provably sound static analysis of ethereum smart contracts. *arXiv preprint arXiv:2005.06227*, 2020.

[SH17] Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. *arXiv preprint arXiv:1702.05511*, 2017.

[SKH18] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level language. *arXiv preprint arXiv:1801.00687*, 2018.

[smt20] SMT-LIB, 2020. Available at `http://smtlib.cs.uiowa.edu/language.shtml`.

[sol19] Solidity. https://solidity.readthedocs.io/, 2019.

[SPY+16] Andrei Stefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 74–91. ACM, 2016.

[SSM20] Clara Schneidewind, Markus Scherer, and Matteo Maffei. The good, the bad and the ugly: Pitfalls and best practices in automated sound static analysis of ethereum smart contracts. In *International Symposium on Leveraging Applications of Formal Methods (ISoLA)*. Springer, 2020.

[ST] Evan Schwartz Stefan Thomas. A Protocol for Interledger Payments. Whitepaper. `https://interledger.org/interledger.pdf`.

[Tam84] Hisao Tamaki. Unfold/fold transformation of logic programs. *Proc. of 2nd ILPC*, pages 127–138, 1984.

[TDDC+18] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical Security Analysis of Smart Contracts. pages 67–82. ACM, January 2018.

[the16] The DAO smart contract, 2016. Available at `http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code`.

[TS+18] Christof Ferreira Torres, Julian Schütte, et al. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, SAC, pages 664–676. ACM, 2018.

[vip] Vyper. Available at `https://github.com/ethereum/vyper`.

[vis] Stress test prepares visanet for the most wonderful time of the year. Blog entry. http://www.visa.com/blogarchives/us/2013/10/10/stress-test-prepares-visanet-for-the-most-wonderful-time-of-the-year/index.html.

[why] Formal verification for solidity contracts. available at https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts.

[Woo14a] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014.

[Woo14b] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.

[Woo16] Gavin Wood. Eip-161: State trie clearing (invariant-preserving alternative). https://eips.ethereum.org/EIPS/eip-161, October 2016. Ethereum Improvement Proposals, no. 161 [Online serial].

[WSX+20] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*, 2020.

[Wui] Peter Wuille. Schnorr Bitcoin Improvement Proposal. https://github.com/sipa/bips/blob/bip-schnorr/bip-schnorr.mediawiki.

[WZ18] Maximilian Wöhrer and Uwe Zdun. Smart contracts: Security patterns in the ethereum ecosystem and solidity. 2018.

[WZS19] Shuai Wang, Chengyu Zhang, and Zhendong Su. Detecting nondeterministic payment bugs in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[YL18] Zheng Yang and Hang Lei. Lolisa: Formal syntax and semantics for a subset of the solidity programming language. *arXiv preprint arXiv:1803.09885*, 2018.

[Zak18] Jakub Zakrzewski. Towards verification of ethereum smart contracts: a formalization of core of solidity. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 229–247. Springer, 2018.

[ZHP+18] Ence Zhou, Song Hua, Bingfeng Pi, Jun Sun, Yashihide Nomura, Kazuhiro Yamashita, and Hidetoshi Kurihara. Security assurance for smart contract. In *New Technologies, Mobility and Security (NTMS), 2018 9th IFIP International Conference on*, pages 1–5. IEEE, 2018.

APPENDIX A

Appendix to Chapter 2

A.1 Formalization

A.1.1 Notations

In the following, we will use $\mathbb{B}$ to denote the set $\{0, 1\}$ of bits and accordingly $\mathbb{B}^x$ for sets of bitstrings of size x. We further let $\mathbb{N}_x$ denote the set of non-negative integers representable by x bits and allow for implicit conversion between those two representations (assuming bitstrings to represent a big-endian encoding of natural numbers). In addition, we will use the notation $[X]$ (resp. $\mathcal{L}(X)$) for arrays (resp. lists) of elements from the set X. We use standard notations for operations on arrays and lists. In particular we write $a\,[pos]$ to access position $pos \in [1, |a| - 1]$ of array $a \in [X]$ and $a[down, up]$ to access the subarray of size $up - down$ from position $down \in [1, |a| - 1]$ to $up \in [1, |a| - 1]$. In case that $down > up$ this operation results in the empty array ϵ. In addition, we write $a_1 \cdot a_2$ for the concatenation of two arrays $a_1, a_2 \in [X]$.

In the following formalization, we will make use of bytearrays $b \in [\mathbb{B}^8]$. To this end, we will assume functions $(\cdot)_{[\mathbb{B}^8]} \in \mathbb{B}^x \to [\mathbb{B}^8]$ and $(\cdot)_{\mathbb{B}} \in [\mathbb{B}^8] \to \mathbb{B}^x$ to chunk bitstrings with size dividable by 8 to bytearrays and vice versa. To denote the zero byte, we write 0^8, and accordingly, for an array of zero bytes of size n, we write $0^{8 \cdot n}$.

For lists, we denote the empty list by ϵ and write $x :: xs$ for placing element $x \in X$ on top of list $xs \in \mathcal{L}(X)$. In addition, we write $xs +\!\!+ ys$ for concatenating lists $xs, ys \in \mathcal{L}(X)$.

We let $\mathcal{A}$ denote the set of 160-bit addresses ($\mathbb{B}^{160}$).

A.1.2 Configurations

The global state of the system is defined by the accounts that are existing and their current state, including their balances and their codes. Formally, the global state is a (partial) mapping from

account addresses to accounts:

$$\sigma \in \Sigma = \mathcal{A} \to (\mathbb{N}_{256} \times \mathbb{N}_{256} \times (\mathbb{B}^{256} \to \mathbb{B}^{256}) \times [\mathbb{B}^8]) \cup \{\bot\}$$

An account (*nonce*, *balance*, *stor*, *code*) is described by the account's balance *balance* $\in \mathbb{N}_{256}$, the state of its persistent storage *stor* $\in \mathbb{B}^{256} \to \mathbb{B}^{256}$, its nonce *nonce* $\in \mathbb{N}_{256}$ and the account's code *code* $\in [\mathbb{B}^8]$.

A configuration S of the execution consists of the stack S of execution states. The call stack S keeps track of the calls made during execution. To this end it consists of execution states of one of the following forms:

- *EXC* denotes an exceptional halting state and can only occur as top element. It expresses that the execution of the current call ended with an exception.
- *HALT*(σ, gas, d, η) denotes regular halting and can only occur as top element. It expresses that the execution of the current call halted in global state $\sigma \in \Sigma$ with transaction effects $\eta \in N$ and with an amount *gas* $\in \mathbb{N}_{256}$ of remaining gas and return data $d \in [\mathbb{B}^8]$
- *REVERT*(gas, d) denotes regular state reverting and can only occur as the top element. It expresses that the execution of the current call was reverted with an amount *gas* $\in \mathbb{N}_{256}$ of remaining gas and return data $d \in [\mathbb{B}^8]$.
- $(\mu, \iota, \sigma, \eta)$ denotes a regular execution state and represents the state of the execution of the current call. A regular execution state includes the local state of the stack machine $\mu \in M$, the execution environment $\iota \in I$ that contains the parameters given to the call and the current global state $\sigma \in \Sigma$, and transaction effects $\eta \in N$.

The reason to make the global state part of the call stack is that it does not change linearly during the execution. In the case of an exception, all effects of the call's execution on the global state are reverted, and the execution continues in the global state of the caller. The same holds for the transaction effects.

In Figure A.1 we give a full grammar for call stacks:

Regular Execution States

In the following we give a detailed description of the components of regular executions state.

Local Machine State The local machine state $\mu \in M = \mathbb{N}_{256} \times \mathbb{N}_{256} \times (\mathbb{B}^{256} \to \mathbb{B}^8) \times \mathbb{N}_{256} \times \mathcal{L}(\mathbb{B}^{256}) \times [\mathbb{B}^8]$ represents the state of the underlying state machine used for execution consists of the following components:

- *gas* $\in \mathbb{N}_{256}$ is the current amount of gas still available for execution;
- *pc* $\in \mathbb{N}_{256}$ is the current program counter;

$$
\begin{array}{rlcll}
\text{Call stacks} & \mathbb{S} & \ni & S & := \; EXC :: S_{plain} \mid HALT(\sigma, g, d, \eta) :: S_{plain} \\
 & & & & \mid REVERT(g, d) :: S_{plain} \mid S_{plain} \\
\text{Plain call stacks} & \mathbb{S}_{plain} & \ni & S_{plain} & := \; (\mu, \iota, \sigma, \eta) :: S_{plain} \\
\text{Machine states} & M & \ni & \mu & := \; (gas, pc, m, i, s, d) \\
\text{Execution environments} & I & \ni & \iota & := \; (actor, input, sender, value, code, f_{mod}) \\
\text{Global states} & \Sigma & \ni & \sigma & \\
\text{Account states} & \mathbb{A} & \ni & acc & := \; (n, b, code, stor) \mid \bot \\
\text{Transaction effects} & N & \ni & \eta & := \; (b, L, S_\dagger) \\
\text{Transaction environments} & \mathcal{T}_{env} & \ni & \Gamma & := \; (o, price, H)
\end{array}
$$

Notations: $d \in [\mathbb{B}^8]$, $g \in \mathbb{N}_{256}$, $\eta \in N$, $o \in \mathcal{A}$, $price \in \mathbb{N}_{256}$, $H \in \mathcal{H}$
$gas \in \mathbb{N}_{256}$, $pc \in \mathbb{N}_{256}$, $m \in \mathbb{B}^{256}, \to \mathbb{B}^8$ $i \in \mathbb{N}_{256}$, $s \in \mathcal{L}(\mathbb{B}^{256})$
$sender \in \mathcal{A}$ $input \in [\mathbb{B}^8]$ $sender \in \mathcal{A}$ $value \in \mathbb{N}_{256}$ $code \in [\mathbb{B}^8]$
$f_{mod} \in \mathbb{B}$ $b \in \mathbb{N}_{256}$ $L \in \mathcal{L}(Ev_{log})$ $S_\dagger \subseteq \mathcal{A}$ $\Sigma = \mathcal{A} \to \mathbb{A}$

Figure A.1: Grammar for calls stacks and transaction environments

- $m \in \mathbb{B}^{256} \to \mathbb{B}^8$ is a mapping from 256-bit words to bytes that represents the local memory;
- $i \in \mathbb{N}_{256}$ is the current number of active words in memory;
- $s \in \mathcal{L}(\mathbb{B}^{256})$ is the local 256-bit word stack of the stack machine;
- $d \in [\mathbb{B}^8]$ is the return data array of the last call

The execution of each internal transaction starts in a fresh machine state, with an empty stack, memory initialized to all zeros, and program counter and active words in memory set to zero. Only the gas is instantiated with the gas value available for the execution.

Execution Environment The execution environment ι of an internal transaction specifies the static parameters of the transaction. It is a tuple of the form $(actor, input, sender, value, code, , f_{mod}) \in I = \mathcal{A} \times [\mathbb{B}^8] \times \mathcal{A} \times \mathbb{N}_{256} \times [\mathbb{B}^8] \times \mathbb{B}$ with the following components:

- $actor \in \mathcal{A}$ is the address of the account currently executing;
- $input \in [\mathbb{B}^8]$ is the data given as an input to the internal transaction;
- $sender \in \mathcal{A}$ is the address of the account that initiated the internal transaction;
- $value \in \mathbb{N}_{256}$ is the value transferred by the internal transaction;
- $code \in [\mathbb{B}^8]$ is the code currently executed;
- $f_{mod} \in \mathbb{B}$ is the flag that indicates whether the current execution may modify the global state.

This information is determined at the beginning of an internal transaction execution, and it can be accessed but not altered during the execution.

Transaction Effects The transaction effects $\eta \in N = \mathbb{N}_{256} \times \mathcal{L}(Ev_{log}) \times \mathcal{P}(\mathcal{A}) \times \mathcal{P}(\mathcal{A})$ collect information on changes that will be applied to the global state after the transaction's execution. They do not effect the code execution itself. In particular, the transaction effects contain the following components:

- $bal_r \in \mathbb{N}_{256}$ is the refund balance that is increased by memory operations and will finally be paid to the transaction's beneficiary
- $L \in \mathcal{L}(Ev_{log})$ is the sequence of log events performed during executions. A log event is a tuple of the address of the currently executing a count, a tuple with zero to four components specified when executing a logging instruction and finally a fraction of the local memory. Consequently, $Ev_{log} = \mathcal{A} \times (\{()\} \cup \mathbb{B}^{256} \cup (\mathbb{N}_{256})^2 \cup (\mathbb{N}_{256})^3 \cup (\mathbb{N}_{256})^4) \times [\mathbb{B}^8]$.
- $S_\dagger \subseteq \mathcal{A}$ is the suicide set that keeps track of the contracts that destroyed themselves (using the SELFDESTRUCT command) during the execution (of the external transaction). These contracts are recorded in $S_\dagger$ and only removed from the global state after the end of the execution.
- $S_\natural \subseteq \mathcal{A}$ is the set of touched accounts that keeps track of the accounts that were involved in the execution. These accounts are recorded in order to remove empty accounts after the execution.

A.1.3 Transaction Environment

The transaction environment represents the static information of the block that the transaction is executed in and the immutable parameters given to the transaction as the gas price or the gas limit. More specifically, the transaction environment $\Gamma \in \mathcal{T}_{env} = \mathcal{A} \times \mathbb{N}_{256} \times \mathcal{H}$ is a tuple of the form $(o, price, H)$ with the following components:

- $o \in \mathcal{A}$ is the address of the account that made the transaction
- $price \in \mathbb{N}_{256}$ denotes the amount of wei that needs to paid for a unit of gas in this transaction
- $H \in \mathcal{H} = \mathbb{N}_{256} \times \mathcal{A} \times \mathbb{N}_{256} \times \mathbb{N}_{256} \times \mathbb{N}_{256} \times \mathbb{N}_{256}$ is the header of the block that the transaction is part. A block header is of the form $(parent, beneficiary, difficulty, number, gaslimit, timestamp)$. Where $parent \in \mathbb{N}_{256}$ identifies the header of the block's parent block, $beneficiary \in \mathcal{A}$ is the address of the beneficiary of the transaction, $difficulty \in \mathbb{N}_{256}$ is a measure of the difficulty of solving the proof of work puzzle required to mine the block, $number \in \mathbb{N}_{256}$ is the number of ancestor blocks, $gaslimit \in \mathbb{N}_{256}$ is the maximum amount of gas that might be consumed when executing the blocks transactions and $timestamp \in \mathbb{N}_{256}$ is the Unix timestamp at the block's inception. Note that this is a simplified version of the block header described in the yellow paper [Woo14b] that only contains those components needed for transaction execution.

A.2 Small-step Semantics

We define a small step relation $\rightarrow$. We write $\Gamma \vDash S \rightarrow S'$ to denote that the call stack $S \in \mathbb{S}$ evolves under the transaction environment $\Gamma \in \mathcal{T}_{env}$ to the call stack $S' \in \mathbb{S}$. The transaction environment contains information concerning the block or transaction the current code is executed in, and that does not change over code execution.

A.2.1 Notations

In order to present the small-step rules concisely, we introduce some notations for accessing and updating the state.

For the global state we use a slightly different notation for accessing and updating. As the global state is a mapping from addresses to account, the account's state can be accessed by applying the address to the global state. For updating we introduce a simplifying notation:

$$\sigma\langle addr \rightarrow s\rangle := \lambda a.\, a = addr\,?\,s\,:\,\sigma(a)$$

For accessing memory fragments we use the following notation:

$$\mathsf{m}\,[o, s] := [\mathsf{m}(o), \mathsf{m}(o+1), \ldots, \mathsf{m}(o+s-1)]$$

Correspondingly, we define updates for memory fragments. Let $o, s \in \mathbb{N}_{256}$ and $v \in [\mathbb{B}^8]$:

$$\mathsf{m}[[o, s] \rightarrow v] := \lambda x.\, (x \geq o \wedge x < o + \mathit{min}\,(s, |v|))\,?\,v\,[x - o]\,:\,\mathsf{m}(x)$$

Similarly to accessing arrays, we write $v[\mathit{down}, \mathit{up}]$ to extract the bitvector's bits from position *down* until position *up* (where we require $\mathit{down} \leq \mathit{up}$). Additionally, we assume a concatenation function for bitvectors and write $b_1 \cdot b_2$ for concatenating bit vectors b_1 and b_2.

Most of the state components used in the formalization of the EVM execution configurations consist of tuples. For the sake of better readability, instead of accessing tuple components using projection, we name the components according to the variable names we used in the description in Section A.1 and use a dot notation for accessing them. To differentiate component names from variable names, we typeset components in sans serifs font. For example, given $\mu \in M$, we write $\mu.\mathsf{gas}$ to access the first component of the tuple μ. Similarly, we use a simple update notation for components. E.g., instead of writing $\mathit{let}\ \mu = (\mathit{gas}, \mathit{pc}, m, i, s)\ \mathit{in}\ (\mathit{gas}, \mathit{pc}+1, m, i, s)$, we write $\mu[\mathsf{pc} \rightarrow \mu.\mathsf{pc} + 1]$. For the case of incrementing or decrementing numerical values we use the usual short cuts $+=$ and $-=$ and would for example write the example shown before as $\mu[\mathsf{pc}\ +=\ 1]$.

As mentioned in section A.1.1, we use the notions of $\mathbb{B}^x$ and $\mathbb{N}_x$ interchangeably as we interpret bitvectos usually as unsigned integers (interpreting the bitvector big-endian). As some operations however are performed on the signed interpretation of the machine words, we assume functions $(\cdot)^- : \mathbb{B}^x \rightarrow \mathit{Int}_x$ and $(\cdot)^- : \mathbb{N}_x \rightarrow \mathit{Int}_x$ that output the signed interpretation of a bitvector or unsigned integer respectively. Note the that Int_x denotes the set of signed integers representable with x bits. Accordingly, we assume a functions $(\cdot)^+ : \mathit{Int}_x \rightarrow \mathbb{B}^x$ and $(\cdot)^+ : \mathit{Int}_x \rightarrow \mathbb{N}_x$ for converting signed integers back to their unsigned interpretation.

A.2.2 Auxiliary Definitions

Accessing bytecode For extracting the command that is currently executed, the instruction at position $\mu.\mathsf{pc}$ of the code code provided in the execution environment needs to be accessed. For the sake of presentation, we define a function doing so:

Definition 10 (Currently executed command)**.** *The currently executed command in the machine state μ and execution environment ι is denoted by $\omega_{\mu,\iota}$ and defined as follows:*

$$\omega_{\mu,\iota} := \begin{cases} \iota.\mathsf{code}\,[\mu.\mathsf{pc}] & \mu.\mathsf{pc} < |\iota.\mathsf{code}| \\ \mathsf{STOP} & \textit{otherwise} \end{cases}$$

All EVM instructions have in common that running out of gas as well as over and underflows of the local machine stack cause an exception. We define a function $\mathit{valid}\,(\cdot,\cdot,\cdot) : \mathbb{N}_{256} \times \mathbb{N}_{256} \times \mathbb{N} \to \mathbb{B}$ that given the available gas, the instruction cost and the new stack size determines whether one of the conditions mentioned above is satisfied. We do not check for stack underflows as this is realized by pattern matching in the individual small step rules.

$$\mathit{valid}\,(g,c,s) := \begin{cases} 1 & g \geq c \wedge s < 1024 \\ 0 & \textit{otherwise} \end{cases}$$

We also write $\mathit{valid}\,(g,c,s)$ for $\mathit{valid}\,(g,c,s) = 1$ and $\neg\mathit{valid}\,(g,c,s)$ for $\mathit{valid}\,(g,c,s) = 0$.

In EVM bytecode potential jump destinations are explicitly marked by the distinct JUMPDEST instruction. Jumps to other destination cause an exception. For simplifying this check, we define the set of valid jump destinations as follows:

Definition 11. *Valid jump destinations [Woo14b]. $D\,(\cdot) : [\mathbb{B}^8] \to \mathcal{P}(\mathbb{N})$ determines the set of valid jump destinations given the code code $\in [\mathbb{B}^8]$, that is being run. It is defined as any position in the code occupied by a JUMPDEST instruction. Formally $D\,(c) = D_H\,(c,0)$, where:*

$$D_H\,(\cdot,\cdot) : [\mathbb{B}^8] \times \mathbb{N} \to \mathcal{P}(\mathbb{N})$$

$$D_H\,(c,i) := \begin{cases} \emptyset & i \geq |c| \\ \{i\} \cup D_H\,(c, N\,(i, c[i])) & c\,[i] = \mathsf{JUMPDEST} \\ D_H\,(c, N\,(i, c\,[i])) & \textit{otherwise} \end{cases}$$

where $N\,(\cdot,\cdot) : \mathbb{N} \times \mathbb{B}^8 \to \mathbb{N}$ is the next valid instruction position in the code, skipping the data of a PUSHn instruction, if any:

$$N\,(i,\omega) := \begin{cases} i+n+1 & \omega = \mathsf{PUSH}n \\ i+n & \textit{otherwise} \end{cases}$$

Memory Consumption The execution tracks the number of active words in memory and charges fees for the memory that is used. The active words in memory are those words that are accessed either for reading or writing. If a command increases the number of active words, it needs to pay accordingly to the number of words that became active.

To model the increasing number of active words in memory we define a memory expansion function as done in [Woo14b] that determines the number of active words in memory given the number of active memory words so far as well as the offset and the size of the memory fraction accessed.

$$M\left(i,o,s\right) := \begin{cases} i & \text{if } s = 0 \\ max(i, \left\lceil \frac{(o+s)}{32} \right\rceil) & \text{otherwise} \end{cases}$$

According to the amount of additional words in memory that are used by the execution of an instruction, additional execution costs are charged. For describing the cost that occur due to memory consumption, we use a function $C_{mem}\left(\cdot,\cdot\right) : \mathbb{N} \times \mathbb{N} \to \mathbb{Z}$ that given the number of active words in memory before and after the command execution, outputs the corresponding costs.

$$C_{mem}\left(aw, aw'\right) := 3 \cdot \left(aw' - aw\right) + \left\lfloor \frac{aw'^2}{512} \right\rfloor - \left\lfloor \frac{aw^2}{512} \right\rfloor$$

Creating New Account Addresses We define a function $newAddress\left(\cdot,\cdot\right) : \mathcal{A} \times \mathbb{N} \to \mathcal{A}$ that given an address and a nonce provides a fresh address.

$$newAddress\left(a,n\right) = Keccak(rlp\left((a, n-1)\right))[96, 255]$$

where $rlp\left(\cdot\right)$ is the RLP encoding function. The RLP encoding is a canonical way of transforming different structures such as tuples to a sequence of bytes. We will not comment on this in detail but refer to the reader to the Ethereum yellow paper [Woo14b].

Note that the $newAddress\left(\cdot,\cdot\right)$ function is assumed to be collision-resistant.

A.2.3 Small-step rules

Binary Stack Operations We start by giving the rules for arithmetic operations. As all of these instructions alter only the local stack and gas and their only difference consists of the operations performed and the (constant) amount of gas computed, we assume a set $Inst_{bin}$ of binary operations and functions $\text{cost}_{bin}(\cdot) : Inst_{bin} \to \mathbb{N}_{256}$ and $\text{fun}_{bin}(\cdot) : Inst_{bin} \to (\mathbb{B}^{256} \times \mathbb{B}^{256} \to \mathbb{B}^{256})$ that map the binary operations to their costs and functionality.

For all binary operations $i_{bin} \in Inst_{bin}$, we create rules of the following form

$$\frac{\begin{array}{c} \omega_{\mu,\iota} = i_{bin} \qquad valid\left(\mu.\mathsf{gas}, \text{cost}_{bin}(i_{bin}), |s|+1\right) \\ \mu.\mathsf{s} = a :: b :: s \qquad \mu' = \mu[\mathsf{s} \to (\text{fun}_{bin}(i_{bin})) :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= \text{cost}_{bin}(i_{bin})] \\ c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code}) \end{array}}{\Gamma \vDash (\mu,\iota,\sigma,\eta) :: S \xrightarrow{i^c_{bin}(a,b)} (\mu',\iota,\sigma,\eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = i_{bin} \qquad (\neg valid\,(\mu.\mathsf{gas}, \mathrm{cost}_{bin}(i_{bin}), |s|+1) \vee |\mu.\mathsf{s}| < 2)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

We define

$$Inst_{bin} := \{\mathsf{ADD}, \mathsf{SUB}, \mathsf{LT}, \mathsf{GT}, \mathsf{EQ}, \mathsf{AND}, \mathsf{OR}, \mathsf{XOR}, \mathsf{SLT}, \mathsf{SGT}, \mathsf{MUL}, \mathsf{DIV}, \mathsf{SDIV}, \mathsf{MOD}, \mathsf{SMOD}, \mathsf{SIGNEXTEND}, \mathsf{BYTE}, \mathsf{SHL}, \mathsf{SHR}, \mathsf{SAR}\}$$

and

$$\mathrm{cost}_{bin}(i_{bin}) = \begin{cases} 3 & i_{bin} \in \{\mathsf{ADD}, \mathsf{SUB}, \mathsf{LT}, \mathsf{GT}, \mathsf{SLT}, \mathsf{SGT}, \mathsf{EQ}, \mathsf{AND}, \mathsf{OR}, \mathsf{XOR}, \mathsf{BYTE} \\ & \quad \mathsf{SHL}, \mathsf{SHR}, \mathsf{SAR}\} \\ 5 & i_{bin} \in \{\mathsf{MUL}, \mathsf{DIV}, \mathsf{SDIV}, \mathsf{MOD}, \mathsf{SMOD}, \mathsf{SIGNEXTEND}\} \end{cases}$$

and

$$\mathrm{fun}_{bin}(i_{bin}) = \begin{cases} \lambda(a,b).\, a + b \mod 2^{256} & i_{bin} = \mathsf{ADD} \\ \lambda(a,b).\, a - b \mod 2^{256} & i_{bin} = \mathsf{SUB} \\ \lambda(a,b).\, a < b\,?\,1 : 0 & i_{bin} = \mathsf{LT} \\ \lambda(a,b).\, a > b\,?\,1 : 0 & i_{bin} = \mathsf{GT} \\ \lambda(a,b).\, a^- < b^-\,?\,1 : 0 & i_{bin} = \mathsf{SLT} \\ \lambda(a,b).\, a^- > b^-\,?\,1 : 0 & i_{bin} = \mathsf{SGT} \\ \lambda(a,b).\, a = b\,?\,1 : 0 & i_{bin} = \mathsf{EQ} \\ \lambda(a,b).\, a \& b & i_{bin} = \mathsf{AND} \\ \lambda(a,b).\, a \| b & i_{bin} = \mathsf{OR} \\ \lambda(a,b).\, a \oplus b & i_{bin} = \mathsf{XOR} \\ \lambda(a,b).\, a \cdot b \mod 2^{256} & i_{bin} = \mathsf{MUL} \\ \lambda(a,b).\, (b=0)\,?\,0 : \lfloor a \div b \rfloor & i_{bin} = \mathsf{DIV} \\ \lambda(a,b).\, (b=0)\,?\,0 : a \mod b & i_{bin} = \mathsf{MOD} \\ \lambda(a,b).\, (b=0)?\,0 : (a = 2^{255} \wedge b^- = -1)?\,2^{256} : & \\ \quad let\ x = a^- \div b^-\ in\ (sign(x) \cdot \lfloor |x| \rfloor)^+ & i_{bin} = \mathsf{SDIV} \\ \lambda(a,b).\, (b=0)\,?\,0 : (sign(a) \cdot |a| \mod |b|)^+ & i_{bin} = \mathsf{SMOD} \\ \lambda(o,b).\, (o \geq 32)\,?\,0 : b[8\cdot o, 8\cdot o + 7] \cdot 0^{248} & i_{bin} = \mathsf{BYTE} \\ \lambda(a,b).\, let\ x = 256 - 8(a+1)\ in & \\ \quad let\ s = b\,[x]\ in\ s^x \cdot b[x, 255] & i_{bin} = \mathsf{SIGNEXTEND} \\ \lambda(a,b).b * 2^a \bmod 2^{256} & i_{bin} = \mathsf{SHL} \\ \lambda(a,b).\lfloor \frac{b}{2^a} \rfloor & i_{bin} = \mathsf{SHR} \\ \lambda(a,b).\lfloor (\frac{b^-}{2^a} \rfloor)^+ & i_{bin} = \mathsf{SAR} \end{cases}$$

where $sign(\cdot) : Int_x \to \{-1, 1\}$ is defined as

$$sign(x) = \begin{cases} 1 & x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

and $\&$, $\|$ and $\oplus$ are bitwise and, or and xor, respectively.

Exceptions to the normal binary operations are the exponentiation (as this instruction uses non-constant costs) and the computation of the Keccack-256 hash.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{EXP} \qquad valid\,(\mu.\mathsf{gas}, c, |s| + 1) \\ \mu.\mathsf{s} = a :: b :: s \qquad c = (b = 0)\,?\,10 \;:\; 10 + 10 * (1 + \lfloor \log_{256} b \rfloor) \qquad x = (a^b) \mod 2^{256} \\ \mu' = \mu[\mathsf{s} \to x :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= c] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{EXP}_c(a,b)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{EXP} \qquad c = (b = 0)\,?\,10 \;:\; 10 + 10 * (1 + \lfloor \log_{256} b \rfloor) \\ \mu.\mathsf{s} = a :: b :: s \qquad \neg valid\,(\mu.\mathsf{gas}, c, |s| + 1)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to EXC :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{SHA3} \\ valid\,(\mu.\mathsf{gas}, c, |s| + 1) \qquad \mu.\mathsf{s} = pos :: size :: s \qquad aw = M\,(\mu.\mathsf{i}, pos, size) \\ c = C_{mem}\,(\mu.\mathsf{i}, aw) + 30 + 6 \cdot \left\lceil \frac{size}{32} \right\rceil \qquad v = \mu.\mathsf{m}\,[pos, pos + size - 1] \qquad h = Keccak(v) \\ \mu' = \mu[\mathsf{s} \to h :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= c][\mathsf{i} \to aw] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{SHA3}_c(pos,size)} (\mu', \iota, \sigma, \eta) :: S}$$

where $Keccak(x)$ is the Keccak-256 hash of x.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{SHA3} \qquad \mu.\mathsf{s} = pos :: size :: s \qquad \mu.\mathsf{s} = pos :: size :: s \qquad aw = M\,(\mu.\mathsf{i}, pos, size) \\ c = C_{mem}\,(\mu.\mathsf{i}, aw) + 30 + 6 \cdot \left\lceil \frac{size}{32} \right\rceil \qquad \mu.\mathsf{s} = a :: b :: s \qquad \neg valid\,(\mu.\mathsf{gas}, c, |s| + 1)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to EXC :: S}$$

$$\frac{(\omega_{\mu,\iota} = \mathsf{EXP} \vee \omega_{\mu,\iota} = \mathsf{SHA3}) \qquad |\mu.\mathsf{s}| < 2}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to EXC :: S}$$

Unary Stack Operations

$$\frac{\omega_{\mu,\iota} = \mathsf{ISZERO} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 3, |s|+1) \qquad \mu.\mathsf{s} = a :: s \qquad x = (a = 0)\,?\,1 : 0 \qquad \mu' = \mu[\mathsf{s} \to x :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 3] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{ISZERO}_c(a)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{NOT} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 3, |s|+1) \qquad \mu.\mathsf{s} = a :: s \qquad x = \neg a \qquad \mu' = \mu[\mathsf{s} \to x :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 3] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{NOT}_c(a)} (\mu', \iota, \sigma, \eta) :: S}$$

where $\neg$ is bitwise negation.

$$\frac{(\omega_{\mu,\iota} = \mathsf{ISZERO} \vee \omega_{\mu,\iota} = \mathsf{NOT}) \qquad (\neg \mathit{valid}\,(\mu.\mathsf{gas}, 3, |s|+1) \vee |\mu.\mathsf{s}| < 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

Ternary Stack Operations

$$\frac{\omega_{\mu,\iota} = \mathsf{ADDMOD} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 8, |s|+1) \qquad \mu.\mathsf{s} = a :: b :: c :: s \qquad x = (c = 0)\,?\,0 : (a + b) \mod c \qquad \mu' = \mu[\mathsf{s} \to x :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 8] \qquad c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{ADDMOD}_{c'}(a,b,c)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{MULMOD} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 8, |s|+1) \qquad \mu.\mathsf{s} = a :: b :: c :: s \qquad x = (c = 0)\,?\,0 : (a \cdot b) \mod c \qquad \mu' = \mu[\mathsf{s} \to x :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 8] \qquad c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{MULMOD}_{c'}(a,b,c)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{(\omega_{\mu,\iota} = \mathsf{ADDMOD} \vee \omega_{\mu,\iota} = \mathsf{MULMOD}) \qquad (\neg \mathit{valid}\,(\mu.\mathsf{gas}, 8, |\mu.\mathsf{s}| - 2) \vee |\mu.\mathsf{s}| < 3)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

Accessing the Execution Environment There are some simple access operations for accessing parts of the execution environment, such as the addresses of the executing account and the caller, the value given to the internal transaction and the sizes of the executed code, and the data given as input to the call.

$$\frac{\omega_{\mu,\iota} = \mathsf{ADDRESS} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}|+1) \qquad \mu' = \mu[\mathsf{s} \to \iota.\mathsf{actor} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{ADDRESS}_c} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{CALLER} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1) \qquad \mu' = \mu[\mathsf{s} \to \iota.\mathsf{sender} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{CALLER}} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{CALLVALUE} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1) \qquad \mu' = \mu[\mathsf{s} \to \iota.\mathsf{value} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{CALLVALUE}_c} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{CODESIZE} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1) \qquad \mu' = \mu[\mathsf{s} \to |\iota.\mathsf{code}| :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{CODESIZE}_c} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{CALLDATASIZE} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1) \qquad \mu' = \mu[\mathsf{s} \to |\iota.\mathsf{input}| :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{CALLDATASIZE}_c} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{(\omega_{\mu,\iota} = \mathsf{ADDRESS} \vee \omega_{\mu,\iota} = \mathsf{CALLER} \vee \omega_{\mu,\iota} = \mathsf{CALLVALUE} \vee \omega_{\mu,\iota} = \mathsf{CODESIZE} \vee \omega_{\mu,\iota} = \mathsf{CALLDATASIZE}) \qquad \neg \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

Accessing the code and the input data in the execution environment is more involved.

The CALLDATALOAD instruction writes the (first 256 bit of) data given as input to the current call to the stack.

$$\frac{\omega_{\mu,\iota} = \mathsf{CALLDATALOAD} \qquad \mu.\mathsf{s} = a :: s \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 3, |\mu.\mathsf{s}|) \qquad k = (|\iota.\mathsf{input}| - a < 0)\,?\,0 : \mathit{min}\,(|\iota.\mathsf{input}| - a, 32) \qquad v' = \iota.\mathsf{input}\,[a, a + k - 1] \qquad v = v' \cdot 0^{256-k\cdot 8} \qquad \mu' = \mu[\mathsf{s} \to v :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 3] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{CALLDATALOAD}_c(a)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{CALLDATALOAD} \qquad \neg \mathit{valid}\,(\mu.\mathsf{gas}, 3, |\mu.\mathsf{s}|)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

The CALLDATACOPY instruction copies the data that was given as input to the current call to the memory.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALLDATACOPY} \qquad \mu.\mathsf{s} = pos_{\mathsf{m}} :: pos_{\mathsf{d}} :: size :: s \\ aw = M\left(\mu.\mathsf{i}, pos_{\mathsf{m}}, size\right) \qquad c = C_{mem}\left(\mu.\mathsf{i}, aw\right) + 3 + 3 \cdot \left\lceil \frac{size}{32} \right\rceil \\ valid\left(\mu.\mathsf{gas}, c, |\mu.\mathsf{s}|\right) \qquad k = (|\iota.\mathsf{input})| - pos_{\mathsf{d}} < 0 \,?\, 0 \,:\, min\left(|\iota.\mathsf{input}| - pos_{\mathsf{d}}, size\right) \\ d' = \iota.\mathsf{input}\left[pos_{\mathsf{d}}, pos_{\mathsf{d}} + k - 1\right] \qquad d = d' \cdot 0^{8 \cdot (size - k)} \\ \mu' = \mu[\mathsf{s} \to s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= c][\mathsf{m} \to \mathsf{m}[[pos_{\mathsf{m}}, pos_{\mathsf{m}} + size - 1] \to d]][\mathsf{i} \to aw] \\ c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{CALLDATACOPY}_{c'}(pos_{\mathsf{m}}, pos_{\mathsf{d}}, size)} (\mu', \iota, \sigma, \eta) :: S}$$

The CODECOPY instruction copies a fraction of the code that is currently executed to the memory.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CODECOPY} \qquad \mu.\mathsf{s} = pos_{\mathsf{m}} :: pos_{code} :: size :: s \\ aw = M\left(\mu.\mathsf{i}, pos_{\mathsf{m}}, size\right) \qquad c = C_{mem}\left(\mu.\mathsf{i}, aw\right) + 3 + 3 \cdot \left\lceil \frac{size}{32} \right\rceil \\ valid\left(\mu.\mathsf{gas}, c, |\mu.\mathsf{s}|\right) \qquad k = (|\iota.\mathsf{code})| - pos_{code} < 0 \,?\, 0 \,:\, min\left(|\iota.code| - pos_{code}, size\right) \\ d' = \iota.\mathsf{code}\left[pos_{code}, pos_{code} + k - 1\right] \qquad d = d' \cdot \mathsf{STOP}^{size - k} \\ \mu' = \mu[\mathsf{s} \to s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= c][\mathsf{m} \to \mathsf{m}[[pos_{\mathsf{m}}, pos_{\mathsf{m}} + size - 1] \to d]][\mathsf{i} \to aw] \\ c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{CODECOPY}_{c}(pos_{\mathsf{m}}, pos_{code}, size)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{(\omega_{\mu,\iota} = \mathsf{CODECOPY} \vee \omega_{\mu,\iota} = \mathsf{CALLDATACOPY}) \qquad |\mu.\mathsf{s}| < 3}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

$$\frac{\begin{array}{c}(\omega_{\mu,\iota} = \mathsf{CODECOPY} \vee \omega_{\mu,\iota} = \mathsf{CALLDATACOPY}) \\ \mu.\mathsf{s} = pos_{\mathsf{m}} :: size :: pos_{code} :: s \qquad aw = M\left(\mu.\mathsf{i}, pos_{\mathsf{m}}, pos_{code}\right) \\ c = C_{mem}\left(\mu.\mathsf{i}, aw\right) + 3 + 3 \cdot \left\lceil \frac{pos_{code}}{32} \right\rceil \qquad \neg valid\left(\mu.\mathsf{gas}, c, |\mu.\mathsf{s}|\right)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

Accessing the Transaction Environment

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{ORIGIN} \qquad valid\left(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1\right) \\ \mu' = \mu[\mathsf{s} \to \Gamma.\mathsf{o} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{ORIGIN}_{c}} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{GASPRICE} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1) \qquad \mu' = \mu[\mathsf{s} \to \Gamma.\mathsf{price} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{GASPRICE}_c} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{(\omega_{\mu,\iota} = \mathsf{ORIGIN} \vee \omega_{\mu,\iota} = \mathsf{GASPRICE}) \qquad \neg \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

The BLOCKHASH command writes the hash of one of the 256 most recently completed block (that is specified on the stack) to the stack:

$$\frac{\omega_{\mu,\iota} = \mathsf{BLOCKHASH} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 20, |\mu.\mathsf{s}|) \qquad \mu.\mathsf{s} = n :: s \qquad h = P\,(\iota.\mathsf{parent}, n, 0) \qquad \mu' = \mu[\mathsf{s} \to h :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 20] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{BLOCKHASH}_c(n)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{BLOCKHASH} \qquad (\neg \mathit{valid}\,(\mu.\mathsf{gas}, 20, |\mu.\mathsf{s}|) \vee |\mu.\mathsf{s}| < 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

where the function $P\,(h, n, a)$ tries to access the block with number n by traversing the block chain starting from h until the counter a reaches the limit of 256 or the genesis block is reached.

$$P\,(h, n, a) := \begin{cases} 0 & n > h.\mathsf{number} \vee a = 256 \vee h = 0 \\ h & n = h.\mathsf{number} \\ P\,(h.\mathsf{parent}, n, a + 1) & \text{otherwise} \end{cases}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{COINBASE} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1) \qquad \mu' = \mu[\mathsf{s} \to (\Gamma.H).\mathsf{beneficiary} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{COINBASE}_c} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{TIMESTAMP} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1) \qquad \mu' = \mu[\mathsf{s} \to (\Gamma.H).\mathsf{timestamp} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{TIMESTAMP}_c} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{NUMBER} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1) \qquad \mu' = \mu[\mathsf{s} \to (\Gamma.H).\mathsf{number} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{NUMBER}_c} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{DIFFICULTY} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1) \qquad \mu' = \mu[\mathsf{s} \to (\Gamma.H).\mathsf{difficulty} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{DIFFICULTY}_c} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{GASLIMIT} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1) \qquad \mu' = \mu[\mathsf{s} \to (\Gamma.H).\mathsf{gaslimit} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{GASLIMIT}_c} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{(\omega_{\mu,\iota} = \mathsf{COINBASE} \vee \omega_{\mu,\iota} = \mathsf{TIMESTAMP} \vee \omega_{\mu,\iota} = \mathsf{NUMBER} \vee \omega_{\mu,\iota} = \mathsf{DIFFICULTY} \vee \omega_{\mu,\iota} = \mathsf{GASLIMIT}) \qquad \neg \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

Accessing the global state

$$\frac{\omega_{\mu,\iota} = \mathsf{BALANCE} \qquad \mu.\mathsf{s} = a :: s \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 400, |s| + 1) \qquad b = (\sigma(a \mod 2^{160}) = (\mathit{nonce}, \mathit{balance}, \mathit{stor}, \mathit{code}))\,?\,\mathit{balance} : 0 \qquad \mu' = \mu[\mathsf{s} \to b :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 400] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{BALANCE}_c(a)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{BALANCE} \qquad (\neg \mathit{valid}\,(\mu.\mathsf{gas}, 400, |\mu.\mathsf{s}|) \vee |\mu.\mathsf{s}| < 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{EXTCODESIZE} \qquad \mu.\mathsf{s} = a :: s \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 700, |s| + 1) \qquad \sigma(a \bmod 2^{160}) \neq \bot \qquad \mathit{size} = |\sigma(a \bmod 2^{160}).\mathsf{code}| \qquad \mu' = \mu[\mathsf{s} \to s :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 700] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{EXTCODESIZE}_c(a)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{EXTCODESIZE} \qquad \mu.\mathsf{s} = a :: s \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 700, |s| + 1) \qquad \sigma(a \bmod 2^{160}) = \bot \qquad \mathit{size} = 0 \qquad \mu' = \mu[\mathsf{s} \to s :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 700] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{EXTCODESIZE}_c(a)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{EXTCODESIZE} \qquad (\neg \mathit{valid}\,(\mu.\mathsf{gas}, 700, |\mu.\mathsf{s}|) \vee |\mu.\mathsf{s}| < 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to \mathit{EXC} :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{EXTCODECOPY} \qquad \mu.\mathsf{s} = a :: \mathit{pos}_{\mathsf{m}} :: \mathit{pos}_{\mathit{code}} :: \mathit{size} :: s \\ \mathit{code} = \sigma(a \mod 2^{160}).\mathsf{code} \qquad \mathit{aw} = M\,(\mu.\mathsf{i}, \mathit{pos}_{\mathsf{m}}, \mathit{size}) \\ c = C_{\mathit{mem}}\,(\mu.\mathsf{i}, \mathit{aw}) + 700 + 3 \cdot \left\lceil \frac{\mathit{size}}{32} \right\rceil \qquad \mathit{valid}\,(\mu.\mathsf{gas}, c, |\mu.\mathsf{s}|) \\ \sigma(a \bmod 2^{160}) \neq \bot \qquad k = (|\mathit{code}|) - \mathit{pos}_{\mathit{code}} < 0\,?\,0 \;:\; \mathit{min}\,(|\mathit{code}| - \mathit{pos}_{\mathit{code}}, \mathit{size}) \\ d' = \mathit{code}\,[\mathit{pos}_{\mathit{code}}, \mathit{pos}_{\mathit{code}} + k - 1] \qquad d = d' \cdot \mathsf{STOP}^{\mathit{size}-k} \\ \mu' = \mu[\mathsf{s} \to s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= c][\mathsf{m} \to \mathsf{m}[[\mathit{pos}_{\mathsf{m}}, \mathit{pos}_{\mathsf{m}} + \mathit{size} - 1] \to d]][\mathsf{i} \to \mathit{aw}] \\ c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{EXTCODECOPY}_{c'}\,(a, \mathit{pos}_{\mathsf{m}}, \mathit{pos}_{\mathit{code}}, \mathit{size})} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{EXTCODECOPY} \qquad \mu.\mathsf{s} = a :: \mathit{pos}_{\mathsf{m}} :: \mathit{pos}_{\mathit{code}} :: \mathit{size} :: s \\ \mathit{aw} = M\,(\mu.\mathsf{i}, \mathit{pos}_{\mathsf{m}}, \mathit{size}) \qquad c = C_{\mathit{mem}}\,(\mu.\mathsf{i}, \mathit{aw}) + 700 + 3 \cdot \left\lceil \frac{\mathit{size}}{32} \right\rceil \\ \mathit{valid}\,(\mu.\mathsf{gas}, c, |\mu.\mathsf{s}|) \qquad \sigma(a \bmod 2^{160}) = \bot \qquad d = \mathsf{STOP}^{\mathit{size}} \\ \mu' = \mu[\mathsf{s} \to s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= c][\mathsf{m} \to \mathsf{m}[[\mathit{pos}_{\mathsf{m}}, \mathit{pos}_{\mathsf{m}} + \mathit{size} - 1] \to d]][\mathsf{i} \to \mathit{aw}] \\ c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{EXTCODECOPY}_{c'}\,(a, \mathit{pos}_{\mathsf{m}}, \mathit{pos}_{\mathit{code}}, \mathit{size})} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{EXTCODECOPY} \qquad |\mu.\mathsf{s}| < 4}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to \mathit{EXC} :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{EXTCODECOPY} \\ \mu.\mathsf{s} = a :: \mathit{pos}_{\mathsf{m}} :: \mathit{size} :: \mathit{pos}_{\mathit{code}} :: s \qquad \mathit{aw} = M\,(\mu.\mathsf{i}, \mathit{pos}_{\mathsf{m}}, \mathit{pos}_{\mathit{code}}) \\ c = C_{\mathit{mem}}\,(\mu.\mathsf{i}, \mathit{aw}) + 700 + 3 \cdot \left\lceil \frac{\mathit{pos}_{\mathit{code}}}{32} \right\rceil \qquad \neg \mathit{valid}\,(\mu.\mathsf{gas}, c, |\mu.\mathsf{s}|)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to \mathit{EXC} :: S}$$

Stack Operations

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{POP} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 2, |s|) \\ \mu.\mathsf{s} = a :: s \qquad \mu' = \mu[\mathsf{s} \to s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{POP}_c} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{POP} \qquad (\neg \mathit{valid}\,(\mu.\mathsf{gas}, 2, |s|) \vee |\mu.\mathsf{s}| < 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow \mathit{EXC} :: S}$$

There are 32 instructions for pushing values to the stack. We summarize the behavior of all these instructions with the following rules by parameterising the instruction with number of following bytes that are pushed to the stack. The $\mathsf{PUSH}n$ (with $m \in [1, 32]$) command pushes the bytes at the next n program counter position to the stack.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{PUSH}x \qquad k = \mathit{min}\,(|\iota.\mathit{code}|, \mu.\mathsf{pc} + x) \\ \mathit{valid}\,(\mu.\mathsf{gas}, 3, |\mu.\mathsf{s}| + 1) \qquad d = \iota.\mathit{code}\,[\mu.\mathsf{pc} + 1, k] \qquad d' = d \cdot 0^{8\cdot(32-(k-\mu.\mathsf{pc}))} \\ \mu' = \mu[\mathsf{s} \rightarrow d' :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= (x+1)][\mathsf{gas} \mathrel{-}= 3] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{PUSH}x_c} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{PUSH}x \qquad \neg \mathit{valid}\,(\mu.\mathsf{gas}, 3, |s| + 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow \mathit{EXC} :: S}$$

The $\mathsf{DUP}n$ instructions (with $n \in [1, 16]$) duplicate the nth stack element:

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{DUP}n \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 3, |\mu.\mathsf{s}| + 1) \qquad \mu.\mathsf{s} = s_1 +\!\!+ (x_n :: s_2) \qquad |s_1| = n - 1 \\ \mu' = \mu[\mathsf{s} \rightarrow x_n :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 3] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{DUP}n_c} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{DUP}n \qquad (\neg \mathit{valid}\,(\mu.\mathsf{gas}, 3, |\mu.\mathsf{s}| + 1) \vee |\mu.\mathsf{s}| < n)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow \mathit{EXC} :: S}$$

The $\mathsf{SWAP}n$ instructions (with $n \in [1, 16]$) swap the first and the $(n-1)$st stack element:

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{SWAP}n \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 3, |\mu.\mathsf{s}|) \qquad \mu.\mathsf{s} = y :: (s_1 +\!\!+ (x_n :: s_2)) \qquad |s_1| = n - 1 \\ \mu' = \mu[\mathsf{s} \rightarrow x_n :: (s_1 +\!\!+ (y :: s_2))][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 3] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{SWAP}n_c} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{SWAP}n \qquad (\neg \mathit{valid}\,(\mu.\mathsf{gas}, 3, |\mu.\mathsf{s}|) \vee |\mu.\mathsf{s}| < n + 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow \mathit{EXC} :: S}$$

Jumps The JUMP command updates the program counter to i (specified in the stack) if i is a valid jump destination.

$$\frac{\begin{array}{ccc}\omega_{\mu,\iota} = \mathsf{JUMP} & \mathit{valid}\,(\mu.\mathsf{gas}, 8, |s|) & \mu.\mathsf{s} = i :: s \\ i \in D\,(\iota.\mathit{code}) & \mu' = \mu[\mathsf{s} \to s][\mathsf{pc} \to i][\mathsf{gas} \mathrel{-}= 8] & c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{JUMP}_c(i)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{JUMP} \qquad \mu.\mathsf{s} = i :: s \qquad (i \notin D\,(\iota.\mathit{code}) \vee \neg \mathit{valid}\,(\mu.\mathsf{gas}, 8, |s|))}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{JUMP} \qquad |\mu.\mathsf{s}| < 1}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

The conditional jump command JUMPI conditionally jumps to position i depending on b. Note that here we corrected a bug in the yellow paper [Woo14b] where it is stated that in case of an invalid jump destination, a conditional jump will always result in an exception. However, in the client implementation, an exception only occurs if the condition on the stack is true. We adapted the semantics here to be in accordance with the client implementations.

$$\frac{\begin{array}{ccc}\omega_{\mu,\iota} = \mathsf{JUMPI} & \mathit{valid}\,(\mu.\mathsf{gas}, 10, |s|) & \mu.\mathsf{s} = i :: b :: s \qquad i \in D\,(\iota.\mathit{code}) \\ b > 0 & \mu' = \mu[\mathsf{s} \to s][\mathsf{pc} \to i][\mathsf{gas} \mathrel{-}= 10] & c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{JUMPI}_c(i,b)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{ccc}\omega_{\mu,\iota} = \mathsf{JUMPI} & \mathit{valid}\,(\mu.\mathsf{gas}, 10, |s|) & \mu.\mathsf{s} = i :: b :: s \\ b = 0 & \mu' = \mu[\mathsf{s} \to s][\mathsf{pc} \to \mu.\mathsf{pc} + 1][\mathsf{gas} \mathrel{-}= 10] & c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{JUMPI}_c(i,b)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{JUMPI} \qquad \mu.\mathsf{s} = i :: b :: s \qquad (b > 0 \ \wedge \ i \notin D\,(\iota.\mathit{code}) \vee \neg \mathit{valid}\,(\mu.\mathsf{gas}, 10, |s|))}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{JUMPI} \qquad |\mu.\mathsf{s}| < 2}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

The JUMPDEST command marks a valid jump destination. It does not trigger any execution, and consequently, the only effect of the command is the increase of the program counter and charging the fee for the command execution.

$$\frac{\omega_{\mu,\iota} = \mathsf{JUMPDEST} \quad \mathit{valid}\,(\mu.\mathsf{gas}, 1, |\mu.\mathsf{s}|) \quad \mu' = \mu[\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1] \quad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{JUMPDEST}_c} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{JUMPDEST} \quad \neg \mathit{valid}\,(\mu.\mathsf{gas}, 1, |\mu.\mathsf{s}|)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

Local Memory Operations The MLOAD command reads a fraction of the local memory specified by a and pushes it to the stack. Note that this increases the number of active words in memory and therefore causes additional cost.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{MLOAD} \quad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 3 \\ \mathit{valid}\,(\mu.\mathsf{gas}, c, |s| + 1) \quad \mu.\mathsf{s} = a :: s \quad v = \mu.\mathsf{m}[a, a + 31] \quad aw = M\,(\mu.\mathsf{i}, a, 32) \\ \mu' = \mu[\mathsf{i} \to aw][\mathsf{s} \to v :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= c] \quad c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{MLOAD}_{c'}(a)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{MLOAD} \quad |\mu.\mathsf{s}| < 1}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{MLOAD} \quad \mu.\mathsf{s} = a :: s \\ aw = M\,(\mu.\mathsf{i}, a, 32) \quad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 3 \quad \neg \mathit{valid}\,(\mu.\mathsf{gas}, c, |s| + 1)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

The MSTORE command writes a value b given at the stack to address a in the local memory. Note that we abuse the update-notation here slightly to update whole intervals of the local memory.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{MSTORE} \\ c = C_{mem}\,(\mu.\mathsf{i}, aw) + 3 \quad \mu.\mathsf{s} = a :: b :: s \quad \mathit{valid}\,(\mu.\mathsf{gas}, c, |s|) \quad aw = M\,(\mu.\mathsf{i}, a, 32) \\ \mu' = \mu[\mathsf{m} \to \mu.\mathsf{m}[[a, a + 31] \to b_{[\mathbb{B}^8]}]][\mathsf{i} \to aw][\mathsf{s} \to s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= c] \\ c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{MSTORE}_{c'}(a,b)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{MSTORE} \qquad \mu.\mathsf{s} = a :: b :: s \qquad aw = M\,(\mu.\mathsf{i}, a, 32) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 3 \qquad \neg valid\,(\mu.\mathsf{gas}, c, |s|)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{MSTORE8} \\ c = C_{mem}\,(\mu.\mathsf{i}, aw) + 3 \quad \mu.\mathsf{s} = a :: b :: s \qquad valid\,(\mu.\mathsf{gas}, c, |s|) \qquad aw = M\,(\mu.\mathsf{i}, a, 1) \\ \mu' = \mu[\mathsf{m} \to \mu.\mathsf{m}[a \to b \mod 256]][\mathsf{i} \to aw][\mathsf{s} \to s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= c] \\ c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{MSTORE8}_{c'}(a,b)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{MSTORE8} \qquad \mu.\mathsf{s} = a :: b :: s \qquad aw = M\,(\mu.\mathsf{i}, a, 1) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 3 \qquad \neg valid\,(\mu.\mathsf{gas}, c, |s|)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

$$\frac{(\omega_{\mu,\iota} = \mathsf{MSTORE} \vee \omega_{\mu,\iota} = \mathsf{MSTORE8}) \qquad |\mu.\mathsf{s}| < 2}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

Persistent Storage Operations The SLOAD command reads the executing account's persistent storage at position a.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{SLOAD} \qquad valid\,(\mu.\mathsf{gas}, 200, |s| + 1) \\ \mu.\mathsf{s} = a :: s \qquad \mu' = \mu[\mathsf{s} \to (\sigma(\iota.addr).\mathsf{stor})(a) :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 200] \\ c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{SLOAD}_c(a)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{SLOAD} \qquad (\neg valid\,(\mu.\mathsf{gas}, 200, |s| + 1) \vee |\mu.\mathsf{s}| < 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

The SSTORE command stores the value b in the executing account's persistent storage at position a.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{SSTORE} \qquad \iota.\mathsf{f}_{\mathsf{mod}} = 1 \qquad c = (b \neq 0 \wedge (\sigma(\iota.addr).\mathsf{stor})(a) = 0)\,?\,20000 : 5000 \\ valid\,(\mu.\mathsf{gas}, c, |s|) \qquad \mu.\mathsf{s} = a :: b :: s \qquad \mu' = \mu[\mathsf{s} \to s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= c] \\ \sigma' = \sigma\langle \iota.addr \to \iota.addr[\mathsf{stor} \to \sigma(\iota.addr).\mathsf{stor}[a \to b]]\rangle \\ r = (b = 0 \wedge (\sigma(\iota.addr).\mathsf{stor})(a) \neq 0)\,?\,15000 : 0 \\ \eta' = \eta[\mathsf{balance} \mathrel{+}= r] \qquad c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{SSTORE}_{c'}(a,b)} (\mu', \iota, \sigma', \eta') :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{SSTORE} \qquad \mu.\mathsf{s} = a :: b :: s \qquad c = (b \neq 0 \wedge (\sigma(\iota.addr).\mathsf{stor})(a) = 0)\ ?\ 20000\ :\ 5000 \qquad \neg valid\,(\mu.\mathsf{gas}, c, |s|)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S\ \to EXC :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{SSTORE} \qquad (|\mu.\mathsf{s}| < 2 \vee \iota.\mathsf{f}_{\mathsf{mod}} = 0)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S\ \to EXC :: S}$$

Accessing the Machine State

$$\frac{\omega_{\mu,\iota} = \mathsf{PC} \qquad valid\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1) \qquad \mu' = \mu[\mathsf{s} \to \mu.\mathsf{pc} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S\ \xrightarrow{\mathsf{PC}_c}\ (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{MSIZE} \qquad valid\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1) \qquad \mu' = \mu[\mathsf{s} \to 32 \cdot \mu.\mathsf{i} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S\ \xrightarrow{\mathsf{MSIZE}_c}\ (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{GAS} \qquad valid\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1) \qquad \mu' = \mu[\mathsf{s} \to \mu.\mathsf{gas} :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S\ \xrightarrow{\mathsf{GAS}_c}\ (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{(\omega_{\mu,\iota} = \mathsf{PC} \vee \omega_{\mu,\iota} = \mathsf{MSIZE} \vee \omega_{\mu,\iota} = \mathsf{GAS}) \qquad \neg valid\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S\ \to EXC :: S}$$

Logging Instructions The logging operation allows appending a new log entry to the log series. The log series keeps track of archived and indexable 'checkpoints' in the execution of Ethereum byte code. The motivation of the log series is to allow external observers to track the program execution. A log entry consists of the address of the currently executing account, up to for 'topics' (specified on the stack), and a fraction of the memory. There are four logging instructions, but as seen before, we describe their effects using common rules parameterizing the instruction by the amount of log information read from the stack.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{LOG}n \qquad \iota.\mathsf{f}_{\mathsf{mod}} = 1 \qquad \mu.\mathsf{s} = pos_{\mathsf{m}} :: size :: (s_1 +\!\!+ s_2) \qquad |s_1| = n \qquad aw = M\,(\mu.\mathsf{i}, pos_{\mathsf{m}}, size) \\ c = C_{mem}\,(\mu.\mathsf{i}, aw) + 375 + 8 \cdot size + n \cdot 375 \qquad valid\,(\mu.\mathsf{gas}, c, |\mu.\mathsf{s}|) \\ \mu' = \mu[\mathsf{s} \to s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= c][\mathsf{i} \to aw] \qquad d = \mu.\mathsf{m}[pos_{\mathsf{m}}, pos_{\mathsf{m}} + size - 1] \\ \eta' = \eta[\mathsf{L} \to \eta.\mathsf{L} +\!\!+ [(\iota.\mathsf{actor}, s_1, d)]] \qquad c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S\ \xrightarrow{\mathsf{LOG}n_{c'}(s_1)}\ (\mu', \iota, \sigma, \eta') :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{LOG}n \\ \mu.\mathsf{s} = pos_{\mathsf{m}} :: size :: (s_1 +\!\!+ s_2) \qquad |s_1| = n \qquad aw = M\,(\mu.\mathsf{i}, pos_{\mathsf{m}}, size) \\ c = C_{mem}\,(\mu.\mathsf{i}, aw) + 375 + 8 \cdot size + n \cdot 375 \qquad \neg valid\,(\mu.\mathsf{gas}, c, |\mu.\mathsf{s}|)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{LOG}n \qquad (|\mu.\mathsf{s}| < n + 2 \vee \iota.\mathsf{f}_{\mathsf{mod}} = 0)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

Halting Instructions The execution of a RETURN command requires reading data from the local memory. Consequently, the cost for memory consumption is charged. Additionally, the read data is recorded in the halting state in order to potentially propagate it to the caller.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{RETURN} \\ \mu.\mathsf{s} = io :: is :: s \qquad aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) \qquad valid\,(\mu.\mathsf{gas}, c, |s|) \\ d = \mu.\mathsf{m}[io, io + is + 1] \qquad g = \mu.\mathsf{gas} - c \qquad c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{RETURN}_{c'}(io,is)} HALT(\sigma, g, d, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{RETURN} \\ \mu.\mathsf{s} = io :: is :: s \qquad aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) \qquad \neg valid\,(\mu.\mathsf{gas}, c, |s|)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{RETURN} \qquad |\mu.\mathsf{s}| < 2}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

The execution of a STOP command halts execution without propagating any data to the caller.

$$\frac{\omega_{\mu,\iota} = \mathsf{STOP} \qquad g = \mu.\mathsf{gas} \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{STOP}_c} HALT(\sigma, g, \epsilon, \eta) :: S}$$

The SELFDESTRUCT instruction deletes the currently executing account. The SELFDESTRUCT command takes one argument from the stack specifying a_{ben} the address of the beneficiary that should get the balance of the suiciding account.

We distinguish the cases where the beneficiary is an existing account and where it still needs to be created. In the latter, an additional fee is charged.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \textsf{SELFDESTRUCT} \qquad \iota.\textsf{f}_{\textsf{mod}} = 1 \qquad \mu.\textsf{s} = a_{ben} :: s \qquad a = a_{ben} \bmod 2^{160} \\ \sigma(a) \neq \bot \qquad c = (\mathit{isEmpty}(\sigma(a)) \ \wedge \ \sigma(\iota.\textsf{actor}).\textsf{balance} > 0)\,?\,37000 \ : \ 5000 \\ \mathit{valid}\,(\mu.\textsf{gas}, [\cancel{5000}]\ c, |s|) \qquad g = \mu.\textsf{gas} - [\cancel{5000}]\ c \\ \sigma' = \sigma\langle a \to \sigma(a)[\textsf{balance} \mathrel{+}= \sigma.(\iota.\textsf{actor}).\textsf{balance}]\rangle\langle \iota.\textsf{actor} \to \sigma(\iota.\textsf{actor})[\textsf{balance} \to 0]\rangle \\ r = (\iota.\textsf{actor} \in \Gamma.S_\dagger)\,?\,24000 \ : \ 0 \\ \eta' = \eta[\textsf{S}_\dagger \to \eta.\textsf{S}_\dagger \cup \{\iota.\textsf{actor}\}][\textsf{balance} \mathrel{+}= r][\textsf{S}_\lightning \to \eta.\textsf{S}_\lightning \cup \{\iota.\textsf{actor}, a\}] \\ c' = (\iota.\textsf{actor}, \sigma(\iota.\textsf{actor}).\textsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\textsf{SELFDESTRUCT}_{c'(a_{ben})}} \mathit{HALT}(\sigma', g, \epsilon, \eta') :: S}$$

Note in particular that due to the order of the update of the global state in case that $a = \iota.\textsf{actor}$ the balance of the executing account will simply be set to 0.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \textsf{SELFDESTRUCT} \qquad \iota.\textsf{f}_{\textsf{mod}} = 1 \qquad \mu.\textsf{s} = a_{ben} :: s \qquad a = a_{ben} \bmod 2^{160} \\ \sigma(a) = \bot \qquad \sigma(\iota.\textsf{actor}).\textsf{balance} > 0 \qquad \mathit{valid}\,(\mu.\textsf{gas}, 37000, |s|) \qquad g = \mu.\textsf{gas} - 37000 \\ \sigma' = \sigma\langle \iota.\textsf{actor} \to \sigma(\iota.\textsf{actor})[\textsf{balance} \to 0]\rangle\langle a \to (0, \sigma(\textsf{actor}).\textsf{balance}, \lambda x.\,0, \epsilon)\rangle \\ r = (\iota.\textsf{actor} \in \Gamma.S_\dagger)\,?\,0 \ : \ 24000 \\ \eta' = \eta[\textsf{S}_\dagger \to \eta.\textsf{S}_\dagger \cup \{\iota.\textsf{actor}\}][\textsf{balance} \mathrel{+}= r][\textsf{S}_\lightning \to \eta.\textsf{S}_\lightning \cup \{\iota.\textsf{actor}, a\}] \\ c' = (\iota.\textsf{actor}, \sigma(\iota.\textsf{actor}).\textsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\textsf{SELFDESTRUCT}_{c'(a_{ben})}} \mathit{HALT}(\sigma', g, \epsilon, \eta') :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \textsf{SELFDESTRUCT} \qquad \mu.\textsf{s} = a_{ben} :: s \qquad a = a_{ben} \bmod 2^{160} \\ c = ((\sigma(a) = \bot \vee \mathit{isEmpty}(\sigma(a))) \ \wedge \ \sigma(\textsf{actor}).\textsf{balance} > 0)\,?\,37000 \ : \ 5000 \\ \neg \mathit{valid}\,(\mu.\textsf{gas}, c, |s|)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \ \to \mathit{EXC} :: S}$$

$$\frac{\omega_{\mu,\iota} = \textsf{SELFDESTRUCT} \qquad (|\mu.\textsf{s}| < 1 \vee \iota.\textsf{f}_{\textsf{mod}} = 0)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \ \to \mathit{EXC} :: S}$$

There is a designated invalid instruction that always causes an exception

$$\frac{\omega_{\mu,\iota} = \textsf{INVALID} \qquad c = (\iota.\textsf{actor}, \sigma(\iota.\textsf{actor}).\textsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\textsf{INVALID}_c} \mathit{EXC} :: S}$$

Calling The CALL command initiates the execution of a (potentially different) account's code. To this end, it gets as parameters the gas g to be spent on the execution, the address *to* of the destination account, the value *va* to be transferred to the destination account. Additionally, a fragment in the local memory containing input data for the called code is specified (by *io* and *is*) and another fragment where the return values of the call are expected (specified by *oo* and *os*). If the recipient *to* exists, the balance of the calling account ι.actor is sufficient to transfer *va* and the call stack limit is not reached yet, the recipient *to* gets the value *va* transferred from the calling account ι.actor. The input data input to the call are read from the local memory and written to the execution environment. Additionally, the execution environment is updated with the information on the originator sender, the owner of the currently executed code actor and the code to be executed (that is the code of the called account). The execution of the called code then starts in the updated execution environment and with an empty machine state.

We introduce some functions for simplifying the cost calculations. First, we introduce a function that calculates the base costs for executing a CALL command (not including costs for memory consumption and the amount of gas given to the callee).

$$C_{base}\,(va, flag) = 700 + (va = 0\,?\,0\ :\ 6500) + (flag = 0\,?\,25000\ :\ 0)$$

The base costs include a fixed amount (700 wei) for calling and additional fees depending on whether ether is transferred or a new account needs to get created.

Next, we introduce the function computing the amount of wei given to a call. This value depends on the amount of ether transferred during the call, on the amount of gas specified on the stack that should be given to the call as well as on the amount of local gas still available to the caller and the fact whether a new contract needs to be created or not.

$$\begin{aligned} C_{gascap}\,(va, flag, g, gas) = \\ &let\ c_{ex} = 700 + (va = 0\,?\,0\ :\ 9000) + (flag = 0\,?\,25000\ :\ 0) \\ &in\ (c_{ex} > gas\,?\,g\ :\ min\,(g, L\,(gas - c_{ex}))) + (va = 0\,?\,0\ :\ 2300) \end{aligned}$$

The information on the transfer value and the existence of the called account influence the number of fixed costs the caller needs to pay for the call independent of the execution of the callee contract. The amount of gas specified on the stack should be given to the callee, but if the local gas runs too low (namely, if the fixed amount to pay already uses too much of the callee's local gas) instead only a predefined fraction of the local gas is given to the call.

We distinguish the cases where a new account needs to get created as the called address does not belong to an existing account and the one where the called account is existing.

First we consider the case where the called account already exists:

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALL} \\ \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \qquad (\iota.\mathsf{f}_{\mathsf{mod}} = 1 \vee va = 0) \qquad to_a = to \mod 2^{160} \\ \sigma(to_a) \neq \bot \qquad |A| + 1 \leq 1024 \qquad \sigma(\iota.\mathsf{actor}).\mathsf{b} \geq va \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ flag = isEmpty(\sigma(to_a))\,?\,0\,:\,1 \qquad c_{call} = C_{gascap}\,(va, [\mp]\,flag, g, \mu.\mathsf{gas}) \\ c = C_{base}\,(va, [\mp]\,flag) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \qquad valid\,(\mu.\mathsf{gas}, c, |s| + 1) \\ \sigma' = \sigma\langle to_a \to \sigma(to_a)[\mathsf{b} \mathrel{+}= va]\rangle\langle \iota.\mathsf{actor} \to \sigma(\iota.\mathsf{actor})[\mathsf{b} \mathrel{-}= va]\rangle \\ d = \mu.\mathsf{m}\,[io, io + is - 1] \qquad \mu' = (c_{call}, 0, \lambda x.\,0, 0, \epsilon, \epsilon) \\ \iota' = \iota[\mathsf{sender} \to \iota.\mathsf{actor}][\mathsf{actor} \to to_a][\mathsf{value} \to va][\mathsf{input} \to d][\mathsf{code} \to \sigma(to_a).\mathsf{code}] \\ \eta' = \eta[\mathsf{S}_{\natural} \to \eta.\mathsf{S}_{\natural} \cup \{\iota.\mathsf{actor}, to_a\}] \qquad c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{CALL}_{c'}(g,to,va,io,is,oo,os)} (\mu', \iota', \sigma', \eta') :: (\mu, \iota, \sigma, \eta) :: S}$$

Next, we consider the case where the called account does not exist. In this case an account with the called address (and the empty code) gets created in executed.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALL} \\ \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \qquad \iota.\mathsf{f}_{\mathsf{mod}} = 1 \qquad va > 0 \qquad to_a = to \mod 2^{160} \\ \sigma(to_a) = \bot \qquad |A| + 1 \leq 1024 \qquad \sigma(\iota.\mathsf{actor}).\mathsf{b} \geq va \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(va, 0, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(va, 0) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ valid\,(\mu.\mathsf{gas}, c, |s| + 1) \qquad \sigma' = \sigma\langle to_a \to (0, va, \lambda x.\,0, \epsilon)\rangle\langle \iota.\mathsf{actor} \to \sigma(\iota.\mathsf{actor})[\mathsf{b} \mathrel{-}= va]\rangle \\ d = \mu.\mathsf{m}\,[io, io + is - 1] \qquad \mu' = (c_{call}, 0, \lambda x.\,0, 0, \epsilon, \epsilon) \\ \iota' = \iota[\mathsf{sender} \to \iota.\mathsf{actor}][\mathsf{actor} \to to_a][\mathsf{value} \to va][\mathsf{input} \to d][\mathsf{code} \to \epsilon] \\ \eta' = \eta[\mathsf{S}_{\natural} \to \eta.\mathsf{S}_{\natural} \cup \{\iota.\mathsf{actor}, to_a\}] \qquad c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{CALL}_{c'}(g,to,va,io,is,oo,os)} (\mu', \iota', \sigma', \eta') :: (\mu, \iota, \sigma, \eta) :: S}$$

If the executing account $\iota.\mathsf{actor}$ does not hold the amount of wei specified to be transferred by the CALL instruction (va) or if the call stack limit of 1024 would be reached by performing the call, the call does not get executed. In the small-step semantics this is modeled by throwing an exception on the callee level.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALL} \\ \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \qquad (\iota.\mathsf{f}_{\mathsf{mod}} = 1 \vee va = 0) \qquad to_a = to \mod 2^{160} \\ flag = (\sigma(to_a) = \bot \vee isEmpty(\sigma(to_a)))\,?\,0\,:\,1 \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(va, flag, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(va, flag) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ valid\,(\mu.\mathsf{gas}, c, |s| + 1) \qquad (va > \sigma((\iota.\mathsf{actor})).\mathsf{balance} \vee |A| + 1 \geq 1024)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to EXC :: (\mu, \iota, \sigma, \eta) :: S}$$

If the execution runs out of gas or the stack limit is exceeded, an exception is thrown:

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALL} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \qquad to_a = to \bmod 2^{160} \\ flag = (\sigma(to_a) = \bot \vee isEmpty(\sigma(to_a)))\,?\,0 : 1 \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(va, flag, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(va, flag) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ (\neg valid\,(\mu.\mathsf{gas}, c, |\mu.\mathsf{s}| - 6) \vee \iota.\mathsf{f}_{\mathsf{mod}} = 0 \wedge va > 0)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow EXC :: S}$$

$$\frac{(\omega_{\mu,\iota} = \mathsf{CALL} \vee \omega_{\mu,\iota} = \mathsf{CALLCODE}) \qquad |\mu.\mathsf{s}| < 7}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow EXC :: S}$$

For returning from a call, there are several options:

1. The execution of the called code ends with RETURN. In this case, the call was successful. The current stack specifies the fragment of the local memory that contains the return value. The return value is copied to the caller's local memory as specified on the caller's stack and the execution proceeds in the global state left by the callee. The caller gets the remaining gas of the caller's execution refunded. To indicate success 1 is written to the caller's stack.

2. The execution of the called code ends with STOP or SELFDESTRUCT. In this case, the return value of the execution is the empty data ϵ that is written to the local memory. This essentially means that nothing is written to the caller's local memory.

3. The execution of the called code ends with an exception. In this case, the remaining arguments are removed from the caller's stack, and instead, 0 is written to the caller's stack. The caller does not get the remaining gas refunded

As the first two cases can be treated analogously, we just need two rules for returning from a call.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALL} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \qquad to_a = to \bmod 2^{160} \\ flag = \sigma.to_a = \bot \vee isEmpty(\sigma(to_a))\,?\,0 : 1 \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(va, flag, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(va, flag) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu' = \mu[\mathsf{i} \rightarrow aw][\mathsf{s} \rightarrow 1 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c][\mathsf{m} \rightarrow \mu.\mathsf{m}[[oo, oo + s - 1] \rightarrow d]][\mathsf{d}_r \rightarrow d]\end{array}}{\Gamma \vDash HALT(\sigma', gas, d, \eta') :: (\mu, \iota, \sigma, \eta) :: S \;\rightarrow (\mu', \iota, \sigma', \eta') :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALL} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \qquad to_a = to \bmod 2^{160} \\ flag = \sigma(to_a) = \bot \vee isEmpty(\sigma(to_a))\,?\,0 : 1 \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(va, flag, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(va, flag) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu' = \mu[\mathsf{i} \rightarrow aw][\mathsf{s} \rightarrow 0 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= c][\mathsf{d}_r \rightarrow \epsilon]\end{array}}{\Gamma \vDash EXC :: (\mu, \iota, \sigma, \eta) :: S \;\rightarrow (\mu', \iota, \sigma, \eta) :: S}$$

The two other instructions for calling (CALLCODE and DELEGATECALL) are similar to CALL.

The CALLCODE instruction only differs in the fact that the control flow is not handed over to the called contract, but only its code is executed in the environment of the calling account. This means that the amount of money transferred is only relevant as a guard for the call but does not need to be actually transferred. In addition, in case that the account whose code should be executed does not exist, this account is not created, but only the empty code is run. However, still, the amount of Ether specified on the stack influences the execution cost.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALLCODE} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \qquad to_a = to \bmod 2^{160} \\ \sigma(to_a) \neq \bot \qquad |A| + 1 \leq 1024 \qquad \sigma(\iota.\mathsf{actor}).\mathsf{b} \geq va \qquad aw = M\left(M\left(\mu.\mathsf{i}, io, is\right), oo, os\right) \\ c_{call} = C_{gascap}\left(va, 1, g, \mu.\mathsf{gas}\right) \qquad c = C_{base}\left(va, 1\right) + C_{mem}\left(\mu.\mathsf{i}, aw\right) + c_{call} \\ valid\left(\mu.\mathsf{gas}, c, |s| + 1\right) \qquad d = \mu.\mathsf{m}\left[io, io + is - 1\right] \qquad \mu' = \left(c_{call}, 0, \lambda x.\, 0, 0, \epsilon, \epsilon\right) \\ \iota' = \iota[\mathsf{sender} \to \iota.\mathsf{actor}][\mathsf{value} \to va][\mathsf{input} \to d][\mathsf{code} \to \sigma(to_a).\mathsf{code}] \\ c' = \left(\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code}\right)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{CALLCODE}_{c'}(g,to,va,io,is,oo,os)} (\mu', \iota', \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALLCODE} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \qquad to_a = to \bmod 2^{160} \\ \sigma(to_a) = \bot \qquad |A| + 1 \leq 1024 \qquad \sigma(\iota.\mathsf{actor}).\mathsf{b} \geq va \qquad aw = M\left(M\left(\mu.\mathsf{i}, io, is\right), oo, os\right) \\ c_{call} = C_{gascap}\left(va, 1, g, \mu.\mathsf{gas}\right) \qquad c = C_{base}\left(va, 1\right) + C_{mem}\left(\mu.\mathsf{i}, aw\right) + c_{call} \\ valid\left(\mu.\mathsf{gas}, c, |s| + 1\right) \qquad d = \mu.\mathsf{m}\left[io, io + is - 1\right] \\ \mu' = \left(c_{call}, 0, \lambda x.\, 0, 0, \epsilon, \epsilon\right) \qquad \iota' = \iota[\mathsf{sender} \to \iota.\mathsf{actor}][\mathsf{value} \to va][\mathsf{input} \to d][\mathsf{code} \to \epsilon] \\ c' = \left(\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code}\right)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{CALLCODE}_{c'}(g,to,va,io,is,oo,os)} (\mu', \iota', \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALLCODE} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \\ to_a = to \bmod 2^{160} \qquad aw = M\left(M\left(\mu.\mathsf{i}, io, is\right), oo, os\right) \\ c_{call} = C_{gascap}\left(va, 1, g, \mu.\mathsf{gas}\right) \qquad c = C_{base}\left(va, 1\right) + C_{mem}\left(\mu.\mathsf{i}, aw\right) + c_{call} \\ valid\left(\mu.\mathsf{gas}, c, |s| + 1\right) \qquad \left(va > \sigma((\iota.\mathsf{actor})).\mathsf{balance} \vee |A| + 1 \geq 1024\right)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to EXC :: (\mu, \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALLCODE} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \\ to_a = to \bmod 2^{160} \qquad aw = M\left(M\left(\mu.\mathsf{i}, io, is\right), oo, os\right) \qquad c_{call} = C_{gascap}\left(va, 1, g, \mu.\mathsf{gas}\right) \\ c = C_{base}\left(va, 1\right) + C_{mem}\left(\mu.\mathsf{i}, aw\right) + c_{call} \qquad \neg valid\left(\mu.\mathsf{gas}, c, |\mu.\mathsf{s}| - 6\right)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to EXC :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALLCODE} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \\ to_a = to \mod 2^{160} \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(va, 1, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(va, 1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu' = \mu[\mathsf{i} \to aw][\mathsf{s} \to 1 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c][\mathsf{m} \to \mu.\mathsf{m}[[oo, oo + s - 1] \to d]][\mathsf{d}_r \to d]\end{array}}{\Gamma \vDash \mathit{HALT}(\sigma', gas, d, \eta') :: (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma', \eta') :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALLCODE} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \\ to_a = to \mod 2^{160} \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(va, 1, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(va, 1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu' = \mu[\mathsf{i} \to aw][\mathsf{s} \to 0 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= c][\mathsf{d}_r \to \epsilon]\end{array}}{\Gamma \vDash \mathit{EXC} :: (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma, \eta) :: S}$$

The DELEGATECALL instruction does not only keep the executing account of the current call but also the transferred value and the sender information. For this reason, the value to be transferred does not need to be specified in the argument in this case. For this reason and because the cost calculation differs (not using the argument value, but the one from the environment), all rules from CALL need to be replicated. Still, the general idea is very similar.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{DELEGATECALL} \qquad \mu.\mathsf{s} = g :: to :: io :: is :: oo :: os :: s \\ to_a = to \bmod 2^{160} \qquad \sigma(to_a) \neq \bot \qquad |A| + 1 \leq 1024 \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(0, 1, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(0, 1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mathit{valid}\,(\mu.\mathsf{gas}, c, |s| + 1) \qquad d = \mu.\mathsf{m}\,[io, io + is - 1] \qquad \mu' = (c_{call}, 0, \lambda x.\,0, 0, \epsilon, \epsilon) \\ \iota' = \iota[\mathsf{input} \to d][\mathsf{code} \to \sigma(to_a).\mathsf{code}] \qquad c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{DELEGATECALL}_{c'}(g, to, io, is, oo, os)} (\mu', \iota', \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{DELEGATECALL} \qquad \mu.\mathsf{s} = g :: to :: io :: is :: oo :: os :: s \\ to_a = to \bmod 2^{160} \qquad \sigma(to_a) = \bot \qquad |A| + 1 \leq 1024 \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(0, 1, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(0, 1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mathit{valid}\,(\mu.\mathsf{gas}, c, |s| + 1) \qquad d = \mu.\mathsf{m}\,[io, io + is - 1] \qquad \mu' = (c_{call}, 0, \lambda x.\,0, 0, \epsilon, \epsilon) \\ \iota' = \iota[\mathsf{input} \to d][\mathsf{code} \to \epsilon] \qquad c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{DELEGATECALL}_{c'}(g, to, io, is, oo, os)} (\mu', \iota', \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{DELEGATECALL} \qquad \mu.\mathsf{s} = g :: to :: io :: is :: oo :: os :: s \\ to_a = to \bmod 2^{160} \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \qquad c_{call} = C_{gascap}\,(0, 1, g, \mu.\mathsf{gas}) \\ c = C_{base}\,(0, 1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, c, |s| + 1) \qquad |A| + 1 \geq 1024\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to \mathit{EXC} :: (\mu, \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{DELEGATECALL} \qquad \mu.\mathsf{s} = g :: to :: io :: is :: oo :: os :: s \\ to_a = to \bmod 2^{160} \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \qquad c_{call} = C_{gascap}\,(0, 1, g, \mu.\mathsf{gas}) \\ c = C_{base}\,(0,1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \qquad \neg valid\,(\mu.\mathsf{gas}, c, |\mu.\mathsf{s}| - 6)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{DELEGATECALL} \qquad |\mu.\mathsf{s}| < 6}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{DELEGATECALL} \qquad \mu.\mathsf{s} = g :: to :: io :: is :: oo :: os :: s \\ to_a = to \bmod 2^{160} \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(0, 1, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(0,1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu' = \mu[\mathsf{i} \to aw][\mathsf{s} \to 1 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c][\mathsf{m} \to \mu.\mathsf{m}[[oo, oo + s - 1] \to d]][\mathsf{d}_r \to d]}{\Gamma \vDash HALT(\sigma', gas, d, \eta') :: (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma', \eta') :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{DELEGATECALL} \qquad \mu.\mathsf{s} = g :: to :: io :: is :: oo :: os :: s \\ to_a = to \bmod 2^{160} \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(0, 1, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(0,1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu' = \mu[\mathsf{i} \to aw][\mathsf{s} \to 0 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= c][\mathsf{d}_r \to \epsilon]}{\Gamma \vDash EXC :: (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma, \eta) :: S}$$

Contract Creation The CREATE command initiates the creation of a new contract. The creation of a new contract is initiated if the call stack limit is not reached yet and if the initial balance *va* that should be initially transferred to the new account does not exceed the balance of the sender (the account owning the currently executed code). In this case, address ρ of the new account is created in dependence of the sender's address $\iota.\mathsf{actor}$ and the sender's address' current nonce incremented by one. If there already exists an account with the address, the balance of this account is transferred to the newly created one. Additionally, the new account gets the specified amount *va* of ether transferred from the sender.

Finally, the execution of the contract starts by executing the initialization code i (i can be found in the local memory $\mu.\mathsf{m}$, its location is specified by the arguments *io* and *is* on the stack). The owner of the initialization code is the newly created account ρ. The owner $\iota.addr$ of the calling code will be recorded as the initiator $\iota.\mathsf{sender}$ of the initialization code execution. The value *va* transferred to the new account is given in the environment parameter $\iota.\mathsf{value}$. The execution starts in the empty machine state with the program counter and the number of active words set to 0, in the empty memory $\lambda x.\, 0$ (the function mapping each number to 0^{256}) and the empty stack ϵ. The original global state σ is recorded in the caller state in order to be able to restore it in the case of an exception in the initiation code execution.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CREATE} \qquad \iota.\mathsf{f}_{\mathsf{mod}} = 1 \qquad \mu.\mathsf{s} = va :: io :: is :: s \qquad aw = M\,(\mu.\mathsf{i}, io, is) \\ c = C_{mem}\,(\mu.\mathsf{i}, aw) + 32000 \qquad valid\,(\mu.\mathsf{gas}, c, |s|+1) \qquad va \leq \sigma(\iota.\mathsf{actor}).\mathsf{balance} \\ |S|+1 \leq 1024 \qquad \rho = newAddress\,(\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{nonce}) \qquad \sigma(\rho) = \bot \\ \sigma' = \sigma\langle\rho \to ([\theta]\ 1, va, \lambda x.\,0, \epsilon)\rangle\langle\iota.\mathsf{actor} \to \sigma(\iota.\mathsf{actor})[\mathsf{balance} \mathrel{-}= va][\mathsf{nonce} \mathrel{+}= 1]\rangle \\ i = \mu.\mathsf{m}\,[io, io + is - 1] \\ \iota' = \iota[\mathsf{sender} \to \iota.\mathsf{actor}][\mathsf{actor} \to \rho][\mathsf{value} \to va][\mathsf{code} \to i][\mathsf{input} \to \epsilon] \\ \mu' = (L\,(\mu.\mathsf{gas} - c), 0, \lambda x.\,0, 0, \epsilon, \epsilon) \\ \eta' = \eta[\mathsf{S}_{\not\iota} \to \eta.\mathsf{S}_{\not\iota} \cup \{\iota.\mathsf{actor}, \rho\}] \qquad c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{CREATE}_{c'}(va,io,is)} (\mu', \iota', \sigma')\eta' :: (\mu, \iota, \sigma)\eta :: S}$$

It should not happen that the newly created address ρ already exists. By making ρ dependent on the active account's address and its nonce (which can be seen as an internal counter on the number of new accounts already created by this account), it should be ensured that the resulting address is unique. However, in practice, the function $newAddress\,(\cdot,\cdot)$ is realized by a hash function that requires dealing with collisions. For the cases where accidentally an existing address is created, the balance of the corresponding account is saved in the newly created one.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CREATE} \qquad \iota.\mathsf{f}_{\mathsf{mod}} = 1 \qquad \mu.\mathsf{s} = va :: io :: is :: s \qquad aw = M\,(\mu.\mathsf{i}, io, is) \\ c = C_{mem}\,(\mu.\mathsf{i}, aw) + 32000 \qquad valid\,(\mu.\mathsf{gas}, c, |s|+1) \qquad va \leq \sigma(\iota.\mathsf{actor}).\mathsf{balance} \\ |S|+1 \leq 1024 \qquad \rho = newAddress\,(\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{nonce}) \\ \sigma(\rho) \neq \bot \qquad \sigma(\rho).\mathsf{nonce} = 0 \qquad \sigma(\rho).\mathsf{code} = \epsilon \qquad b = \sigma(\rho).\mathsf{balance} + va \\ \sigma' = \sigma\langle\rho \to ([\theta]\ 1, b, \lambda x.\,0, \epsilon)\rangle\langle\iota.\mathsf{actor} \to \sigma(\iota.\mathsf{actor})[\mathsf{balance} \mathrel{-}= va][\mathsf{nonce} \mathrel{+}= 1]\rangle \\ i = \mu.\mathsf{m}\,[io, io + is - 1] \\ \iota' = \iota[\mathsf{sender} \to \iota.\mathsf{actor}][\mathsf{actor} \to \rho][\mathsf{value} \to va][\mathsf{code} \to i][\mathsf{input} \to \epsilon] \\ \mu' = (L\,(\mu.\mathsf{gas} - c), 0, \lambda x.\,0, 0, \epsilon, \epsilon) \\ \eta' = \eta[\mathsf{S}_{\not\iota} \to \eta.\mathsf{S}_{\not\iota} \cup \{\iota.\mathsf{actor}, \rho\}] \qquad c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{CREATE}_{c'}(va,io,is)} (\mu', \iota', \sigma', \eta') :: (\mu, \iota, \sigma, \eta) :: S}$$

Similarly to the CALL case, the execution of the CREATE instruction can fail at call time in the case that either the value va to be transferred to the newly created account exceeds the calling account's balance or if the call stack limit is reached.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CREATE} \qquad \iota.\mathsf{f}_{\mathsf{mod}} = 1 \\ \mu.\mathsf{s} = va :: io :: is :: s \qquad aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 32000 \\ valid\,(\mu.\mathsf{gas}, c, |s|+1) \qquad (va > \sigma(\iota.\mathsf{actor}).\mathsf{balance} \vee |S|+1 > 1024)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to EXC :: (\mu, \iota, \sigma, \eta) :: S}$$

In addition the usual out-of-gas exception and violations of the stack limits need to be considered:

$$\frac{\omega_{\mu,\iota} = \mathsf{CREATE} \qquad \mu.\mathsf{s} = va :: io :: is :: s \qquad aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 32000 \qquad \neg valid\,(\mu.\mathsf{gas}, c, |s| + 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{CREATE} \qquad (\mu.\mathsf{s} < 3 \vee \iota.\mathsf{f}_{\mathsf{mod}} = 0)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

To return from contract creation we need to consider different cases:

1. The initialization code ends with a RETURN. In this case contract creation was successful. The return value specifies the code of the new contract. This code will be executed when the contract is called later on. To indicate success and to make the newly created contract accessible to the caller, the address of the new contract account is written to the stack. The caller of the contract creation needs to proceed with the remaining gas from the contract creation and additionally needs to pay a final contract creation cost depending on the length of the contract body code.

2. The initialization code ends with STOP or SELFDESTRUCT. In this case, contract creation was theoretically successful, but no practical, usable contract was created as calls to this contract do not cause code to be executed. Nevertheless, the final contract creation cost needs to be paid.

3. The initialization code causes an exception. In this case, the contract creation was not successful. The former global state is restored, and therefore, all side effects of the contract creation are deleted. To indicate the contract creation's failure, the number 0 is written to the stack of the caller. Additionally, all gas of the caller state is deleted.

Cases number one and two result in regular halting of the callee. The command-specific changes affecting the global state, the remaining gas, and the output data are recorded in the halting state. In the case of contract creation, a final fee is charged that depends on the size of the return data. If the gas remaining from the execution of the initialization code is not sufficient to pay the additional fee, an exception occurs.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CREATE} \qquad \mu.\mathsf{s} = va :: io :: is :: s \\ aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 32000 \qquad c_{final} = 200 \cdot |d| \\ |d| \leq 24576 \qquad gas \geq c_{final} \qquad \rho = newAddress\,(\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{nonce}) \qquad \sigma'(\rho) \neq \bot \\ \neg isEmpty(\sigma'(\rho)) \qquad \mu' = \mu[\mathsf{s} \to \rho :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c - c_{final}][\mathsf{i} \to aw][\mathsf{d}_r \to \epsilon] \\ \sigma'' = \sigma'\langle \rho \to \sigma'(\rho)[\mathsf{code} \to d]\rangle\end{array}}{\Gamma \vDash HALT(\sigma', gas, d, \eta') :: (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma'', \eta') :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{CREATE} \qquad c_{final} = 200 \cdot |d| \qquad (gas < c_{final} \vee |d| > 24576)}{\Gamma \vDash \mathit{HALT}(\sigma', gas, d, \eta') :: (\mu, \iota, \sigma, \eta) :: S \;\rightarrow \mathit{EXC} :: (\mu, \iota, \sigma, \eta) :: S :: S}$$

In the case of exceptional halting of the callee, as in the CALL case, the remaining gas is not refunded and the global state as well as the transaction effects are reverted.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CREATE} \\ \mu.\mathsf{s} = va :: io :: is :: s \qquad aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 32000 \\ \mu' = \mu[\mathsf{s} \rightarrow 0 :: s][\mathsf{pc} \mathrel{-}= 1][\mathsf{gas} \mathrel{+}= c][\mathsf{i} \rightarrow aw][\mathsf{d}_r \rightarrow \epsilon]\end{array}}{\Gamma \vDash \mathit{EXC} :: (\mu, \iota, \sigma, \eta) :: S \;\rightarrow (\mu', \iota, \sigma, \eta) :: S}$$

A.3 EVM Changes

The semantics originally presented in this work refers to the semantics of smart contracts as originally implemented and defined in the yellow paper [Woo14b] after the Homestead hard fork. Since then, several hard forks of the Ethereum blockchain have been performed that also affected the semantics of EVM bytecode. In the following, we will give a summary of these changes and their implications for the semantics. We have marked the changes to the rules in the previous section in blue. Since several of the described changes interfere with each other, we also mark such inter-dependencies in the newly introduced rules in blue – even if the changes happened consecutively.

A.3.1 Changed Behavior

Account Garbage Collection In the original version of the EVM semantics, whenever a non-existent contract was accessed (e.g., being the target of a contract call), and it did not exist before, it was created in an ad-hoc manner. While in some cases (i.e., when money was transferred to such an account) such behavior was inevitable to prevent the loss of money in the system, in the absence of money transfer, this behavior resulted in the creation of dummy accounts without balance, code or storage value.

For this reason, it was decided to establish the invariant that non-existing accounts and such accounts that actually have no code and a balance and a nonce of 0 should be considered the same. The latters are also called *empty* accounts. Empty and non-existent accounts are summarized under the notion of *dead* contracts.

We formally define the notion of emptiness by the following function $\mathit{isEmpty}(\cdot)$ that checks whether an account is empty.

$$\mathit{isEmpty}(\cdot) \in \mathbb{A} \rightarrow \mathbb{B}$$

$$\mathit{isEmpty}(\mathit{account}) := \begin{cases} 1 & \mathit{account}.\mathsf{b} = 0 \;\wedge\; \mathit{account}.\mathsf{n} = 0 \;\wedge\; \mathit{account}.\mathsf{code} = \epsilon \\ 0 & \text{otherwise} \end{cases}$$

For enforcing this invariant, it was decided that ad-hoc creation of empty accounts should be prevented. In particular, this meant that whenever an empty contract would created it instead should be changed to be non-existent.

By keeping the global state free from empty accounts, the global state of the Ethereum system can be represented with less storage space which is an advantage for the users of the system.

Still, due to the old version of the semantics still many empty accounts are populating the system. To mitigate this issue, another change to the semantics was introduced, namely that all accounts that are involved in (potentially state changing) interactions of a contract are tracked and after a transaction's execution checked for being empty and, if so removed from the global state. In this way, existing empty accounts (when still being involved in interactions) can be eliminated.

To account for these changes, several modifications need to be made:

1. All rules that potentially might create empty accounts need to be revised to prevent these cases.
2. Whenever it is checked for the existence of the account, it should also be checked for its non-emptiness (to ensure that these two account forms are treated equivalently)
3. Accounts involved in state-changing interactions need to be recorded. To this end, a set of *touched contracts* is introduced in the transaction effects. The elements from this set will then be checked for emptiness and (similarly to the accounts in the suicide set) deleted after transaction execution.

These changes were implemented as part of the Spurious dragon hard fork and are described in [Woo16].

We first devise the rules for preventing the creation of empty contracts.

In the case of the CREATE instruction, it needs to be checked whether the new contract to be deployed would be such an empty contract. A contract is only created if it is not empty at the point of finalizing the creation. Otherwise, it stays empty. This accounts for the case that the contract was self-destructed during execution.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CREATE} \qquad \mu.\mathsf{s} = va :: io :: is :: s \qquad aw = M\,(\mu.\mathsf{i}, io, is) \\ c = C_{mem}\,(\mu.\mathsf{i}, aw) + 32000 \qquad c_{final} = 200 \cdot |d| \qquad |d| \leq 24576 \qquad gas \geq c_{final} \\ \rho = newAddress\,(\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{nonce}) \qquad (\sigma'(\rho) = \bot \ \vee \ isEmpty(\sigma'(\rho))) \\ \mu' = \mu[\mathsf{s} \to \rho :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c - c_{final}][\mathsf{i} \to aw][\mathsf{d}_r \to \epsilon] \\ \sigma'' = \sigma'\langle \rho \to \bot \rangle \end{array}}{\Gamma \vDash HALT(\sigma', gas, d, \eta') :: (\mu, \iota, \sigma, \eta) :: S \ \to (\mu', \iota, \sigma'', \eta') :: S}$$

For the case of call instructions, it needs to be distinguished between the cases where the money is transferred and those where this is not the case. In the case of (positive) money transfers, the old semantics stays in place. However, in the alternative cases, no new account is created. We present the corresponding rules here.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALL} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \qquad va = 0 \\ to_a = to \bmod 2^{160} \qquad \sigma(to_a) = \bot \qquad |A| + 1 \leq 1024 \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(va, 1, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(va, 1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ valid\,(\mu.\mathsf{gas}, c, |s| + 1) \qquad d = \mu.\mathsf{m}\,[io, io + is - 1] \qquad \mu' = (c_{call}, 0, \lambda x.\,0, 0, \epsilon, \epsilon) \\ \iota' = \iota[\mathsf{sender} \to \iota.\mathsf{actor}][\mathsf{actor} \to to_a][\mathsf{value} \to va][\mathsf{input} \to d][\mathsf{code} \to \epsilon] \\ \eta' = \eta[\mathsf{S}_{\not t} \to \eta.\mathsf{S}_{\not t} \cup \{\iota.\mathsf{actor}, to_a\}] \qquad c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{CALL}_{c'}(g,to,va,io,is,oo,os)} (\mu', \iota', \sigma, \eta') :: (\mu, \iota, \sigma, \eta) :: S}$$

Note that CALLCODE and DELEGATECALL already had the right behavior before (not to create non-existing accounts). This is as both these opcodes do not transfer money but only use the code of the callee. In case that the callee should not exist, they simply execute the empty code. Due to this, there was never a pressing reason to create new accounts (to transfer money to them), and hence these opcodes already implemented the correct behavior.

When formulating the rules for calls, one needs to consider another subtlety: The gas cost charged used to depend on the existence of accounts. Now the cases of non-existence need to be treated the same and hence should charge the same gas cost. Since the idea is that in the case of 0-value transfers to dead contracts, no contract should also be created no creation fee will be charged in these cases. We needed to adapt this in the case where the recipient account exists: In case that this account should be empty still the creation cost should be charged (even though the contract will not be created).

The same argument used for calls also applies to the SELFDESTRUCT instruction. We introduce the rule for the case that the beneficiary is a non-existent contract. This contract will then not be created.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{SELFDESTRUCT} \qquad \iota.\mathsf{f}_{\mathsf{mod}} = 1 \\ \mu.\mathsf{s} = a_{ben} :: s \qquad a = a_{ben} \bmod 2^{160} \qquad \sigma(a) = \bot \qquad \sigma(\iota.\mathsf{actor}).\mathsf{balance} = 0 \\ valid\,(\mu.\mathsf{gas}, 5000, |s|) \qquad g = \mu.\mathsf{gas} - 5000 \qquad r = (\iota.\mathsf{actor} \in \Gamma.S_\dagger)\,?\,0 \,:\, 24000 \\ \eta' = \eta[\mathsf{S}_\dagger \to \eta.\mathsf{S}_\dagger \cup \{\iota.\mathsf{actor}\}][\mathsf{balance} \mathrel{+}= r][\mathsf{S}_{\not t} \to \eta.\mathsf{S}_{\not t} \cup \{\iota.\mathsf{actor}, a\}] \\ c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{SELFDESTRUCT}_{c'}(a_{ben})} HALT(\sigma, g, \epsilon, \eta') :: S}$$

Together with these changes, also a limit on the account code size was introduced [But16]. This change ensures that the creation of a contract with a size of more than 24576 fails before getting deployed. We added and marked the corresponding change in the rules of the CREATE opcode.

Absence of Overwriting in Case of Hash Collisions In the original semantics, it was possible that in the case of a hash collision, an existing contract would be overwritten when performing a CREATE transaction. This behavior was changed to a contract creation failing in case that the created address should belong to an account that already exists.

$$\frac{\omega_{\mu,\iota} = \mathsf{CREATE} \qquad \iota.\mathsf{f_{mod}} = 1 \qquad \rho = \mathit{newAddress}\,(\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{nonce}) \qquad \sigma(\rho) \neq \bot \qquad (\sigma(\rho).\mathsf{nonce} > 0 \vee \sigma(\rho).\mathsf{code} \neq \epsilon)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to \mathit{EXC} :: (\mu, \iota, \sigma, \eta) :: S}$$

Still, in cases where the account exists, but it is considered empty since it has no code and a nonce of 0 then still the old behavior applies. We have marked this restriction in the original rule.

A.3.2 New Opcodes

STATICCALL In addition to CALL, CALLCODE, and DELEGATECALL, STATICCALL [BR17] was introduced as part of the Byzantium hard fork as the fourth opcode for calling other contracts. In contrast to the other call types, STATICCALL does not allow for money transfers and generally requires that the global state is not changed during the execution of the initiated internal transaction. Whenever during the internal transaction, an attempt to changing the state is made, then the execution exceptionally halts. To model this behavior, we need to add an additional component to the execution environment that indicates whether the current transaction is allowed to perform state modifications. Testing this flag then needs to become a precondition for the successful execution of the opcodes SSTORE, SELFDESTRUCT, LOGx and CALL (given that a positive value is transferred with the calls). We marked these additional checks in the corresponding small-step rules.

We give here the small-step rules for the STATICCALL instruction itself:

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{STATICCALL} \qquad \mu.\mathsf{s} = g :: to :: io :: is :: oo :: os :: s \qquad to_a = to \mod 2^{160} \\ \sigma(to_a) \neq \bot \qquad |A| + 1 \leq 1024 \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(0, 1, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(0, 1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mathit{valid}\,(\mu.\mathsf{gas}, c, |s| + 1) \qquad d = \mu.\mathsf{m}\,[io, io + is - 1] \qquad \mu' = (c_{call}, 0, \lambda x.\,0, 0, \epsilon, \epsilon) \\ \iota' = \iota[\mathsf{sender} \to \iota.\mathsf{actor}][\mathsf{actor} \to to_a][\mathsf{value} \to 0][\mathsf{input} \to d][\mathsf{code} \to \sigma(to_a).\mathsf{code}][\mathsf{f_{mod}} \to 0] \\ c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{STATICCALL}_{c'}(g,to,io,is,oo,os)} (\mu', \iota', \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S}$$

The following rule describes the case where the called account does not exist. In this case, an account with the called address (and the empty code) gets created in executed.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{STATICCALL} \qquad \mu.\mathsf{s} = g :: to :: io :: is :: oo :: os :: s \\ to_a = to \mod 2^{160} \qquad \sigma(to_a) = \bot \qquad |A| + 1 \leq 1024 \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(0, 1, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(0, 1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mathit{valid}\,(\mu.\mathsf{gas}, c, |s| + 1) \qquad d = \mu.\mathsf{m}\,[io, io + is - 1] \qquad \mu' = (c_{call}, 0, \lambda x.\,0, 0, \epsilon, \epsilon) \\ \iota' = \iota[\mathsf{sender} \to \iota.\mathsf{actor}][\mathsf{actor} \to to_a][\mathsf{value} \to 0][\mathsf{input} \to d][\mathsf{code} \to \epsilon][\mathsf{f_{mod}} \to 0] \\ c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{STATICCALL}_{c'}(g,to,io,is,oo,os)} (\mu', \iota', \sigma, \eta) :: (\mu, \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{STATICCALL} \qquad \mu.\mathsf{s} = g :: to :: io :: is :: oo :: os :: s \\ to_a = to \bmod 2^{160} \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \qquad c_{call} = C_{gascap}\,(0, 1, g, \mu.\mathsf{gas}) \\ c = C_{base}\,(0,1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \qquad valid\,(\mu.\mathsf{gas}, c, |s| + 1) \qquad |A| + 1 \geq 1024\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow EXC :: (\mu, \iota, \sigma, \eta) :: S}$$

If the execution runs out of gas or the stack limit is exceeded, an exception is thrown:

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{STATICCALL} \qquad \mu.\mathsf{s} = g :: to :: io :: is :: oo :: os :: s \\ to_a = to \bmod 2^{160} \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \qquad c_{call} = C_{gascap}\,(0, 1, g, \mu.\mathsf{gas}) \\ c = C_{base}\,(0,1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \qquad \neg valid\,(\mu.\mathsf{gas}, c, |\mu.\mathsf{s}| - 5)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow EXC :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{STATICCALL} \qquad |\mu.\mathsf{s}| < 6}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow EXC :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{STATICCALL} \qquad \mu.\mathsf{s} = g :: to :: io :: is :: oo :: os :: s \\ to_a = to \mod 2^{160} \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(0, 1, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(0,1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu' = \mu[\mathsf{i} \rightarrow aw][\mathsf{s} \rightarrow 1 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c][\mathsf{m} \rightarrow \mu.\mathsf{m}[[oo, oo + s - 1] \rightarrow d]][\mathsf{d}_r \rightarrow d]\end{array}}{\Gamma \vDash HALT(\sigma', gas, d, \eta') :: (\mu, \iota, \sigma, \eta) :: S \;\rightarrow (\mu', \iota, \sigma', \eta') :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{STATICCALL} \qquad \mu.\mathsf{s} = g :: to :: io :: is :: oo :: os :: s \\ to_a = to \mod 2^{160} \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(0, 1, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(0,1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu' = \mu[\mathsf{i} \rightarrow aw][\mathsf{s} \rightarrow 0 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= c][\mathsf{d}_r \rightarrow \epsilon]\end{array}}{\Gamma \vDash EXC :: (\mu, \iota, \sigma, \eta) :: S \;\rightarrow (\mu', \iota, \sigma, \eta) :: S}$$

RETURNDATACOPY and RETURNDATASIZE The original implementation of the EVM had the limitation that the return data needed to be copied to the local memory at the point of returning to a predefined memory fragment hence not allowing for accessing arbitrary-sized return data. To solve this, two new opcodes RETURNDATACOPY and RETURNDATASIZE [Rei17] were introduced in the Byzantium hard fork to access the data returned by the last call. To realize these opcodes, we need to add another element to the machine state that represents the return data, and that is set upon returning from a transaction initiating instruction with the data given in the halt state. The RETURNDATACOPY and RETURNDATASIZE instructions can then be simply realized by accessing this component of the machine state as done by other data accessing opcodes. We marked the changes needed to initialize and update the return data component in the corresponding rules. Here we give the instructions for the execution of RETURNDATACOPY and RETURNDATASIZE.

$$\frac{\omega_{\mu,\iota} = \mathsf{RETURNDATASIZE} \qquad \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1) \qquad \mu' = \mu[\mathsf{s} \to |\mu.\mathsf{d}_r| :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 2] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{RETURNDATASIZE}_c} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{RETURNDATASIZE} \neg \mathit{valid}\,(\mu.\mathsf{gas}, 2, |\mu.\mathsf{s}| + 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{RETURNDATACOPY} \qquad \mu.\mathsf{s} = \mathit{pos}_\mathsf{m} :: \mathit{pos}_\mathsf{d} :: \mathit{size} :: s \qquad \mathit{aw} = M\,(\mu.\mathsf{i}, \mathit{pos}_\mathsf{m}, \mathit{size}) \\ c = C_{\mathit{mem}}\,(\mu.\mathsf{i}, \mathit{aw}) + 3 + 3 \cdot \left\lceil \frac{\mathit{size}}{32} \right\rceil \qquad \mathit{valid}\,(\mu.\mathsf{gas}, c, |\mu.\mathsf{s}|) \qquad \mathit{pos}_\mathsf{d} + \mathit{size} \leq |\mu.\mathsf{d}_r)| \\ k = \mathit{min}\,(|\mu.\mathsf{d}_r| - \mathit{pos}_\mathsf{d}, \mathit{size}) \qquad d' = \mu.\mathsf{d}_r\,[\mathit{pos}_\mathsf{d}, \mathit{pos}_\mathsf{d} + k - 1] \qquad d = d' \cdot 0^{8 \cdot (\mathit{size} - k)} \\ \mu' = \mu[\mathsf{s} \to s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= c][\mathsf{m} \to \mathsf{m}[[\mathit{pos}_\mathsf{m}, \mathit{pos}_\mathsf{m} + \mathit{size} - 1] \to d]][\mathsf{i} \to \mathit{aw}] \\ c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{RETURNDATACOPY}_{c'}(\mathit{pos}_\mathsf{m}, \mathit{pos}_\mathsf{d}, \mathit{size})} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{RETURNDATACOPY} \qquad |\mu.\mathsf{s}| < 3}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{RETURNDATACOPY} \\ \mu.\mathsf{s} = \mathit{pos}_\mathsf{m} :: \mathit{size} :: \mathit{pos}_\mathsf{d} :: s \qquad \mathit{aw} = M\,(\mu.\mathsf{i}, \mathit{pos}_\mathsf{m}, \mathit{pos}_{\mathit{code}}) \\ c = C_{\mathit{mem}}\,(\mu.\mathsf{i}, \mathit{aw}) + 3 + 3 \cdot \left\lceil \frac{\mathit{pos}_{\mathit{code}}}{32} \right\rceil \qquad (\neg \mathit{valid}\,(\mu.\mathsf{gas}, c, |\mu.\mathsf{s}|) \vee \mathit{pos}_\mathsf{d} + \mathit{size} > |\mu.\mathsf{d}_r)|)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to \mathit{EXC} :: S}$$

REVERT The REVERT opcode [BM17] that was added in the Byzantium hard fork introduces a third form of returning from an (internal) transaction instead of failing or regularly halting the execution of the REVERT opcode allows for purposefully failing while still returning the remaining gas and potentially return data. This, in particular, allows for communicating error messages to the caller via the return data. We can model this in the small-step semantics by a new kind of final state which, similar to the halt states, carries return data and the remaining gas from the execution.

We give the rules for this new opcode here.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{REVERT} \\ \mu.\mathsf{s} = \mathit{io} :: \mathit{is} :: s \qquad \mathit{aw} = M\,(\mu.\mathsf{i}, \mathit{io}, \mathit{is}) \qquad c = C_{\mathit{mem}}\,(\mu.\mathsf{i}, \mathit{aw}) \qquad \mathit{valid}\,(\mu.\mathsf{gas}, c, |s|) \\ d = \mu.\mathsf{m}[\mathit{io}, \mathit{io} + \mathit{is} + 1] \qquad g = \mu.\mathsf{gas} - c \qquad c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{REVERT}_{c'}(\mathit{io}, \mathit{is})} \mathit{REVERT}(g, d) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{REVERT} \qquad \mu.\mathsf{s} = io :: is :: s \qquad aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) \qquad \neg valid\,(\mu.\mathsf{gas}, c, |s|)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{REVERT} \qquad |\mu.\mathsf{s}| < 2}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to EXC :: S}$$

We need to specify the corresponding rules for all transaction initiating instructions to complete an internal transaction that ended up in a revert state.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALL} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \qquad to_a = to \bmod 2^{160} \\ flag = \sigma.to_a = \bot \vee isEmpty(\sigma(to_a))\,?\,0 : 1 \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(va, flag, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(va, flag) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu' = \mu[\mathsf{i} \to aw][\mathsf{s} \to 0 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c][\mathsf{m} \to \mu.\mathsf{m}[[oo, oo + s - 1] \to d]][\mathsf{d}_r \to d]\end{array}}{\Gamma \vDash REVERT(gas, d) :: (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALLCODE} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \\ to_a = to \mod 2^{160} \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(va, 1, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(va, 1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu' = \mu[\mathsf{i} \to aw][\mathsf{s} \to 1 :: s][\mathsf{pc} \mathrel{+}= 0][\mathsf{gas} \mathrel{+}= gas - c][\mathsf{m} \to \mu.\mathsf{m}[[oo, oo + s - 1] \to d]][\mathsf{d}_r \to d]\end{array}}{\Gamma \vDash REVERT(gas, d) :: (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{DELEGATECALL} \qquad \mu.\mathsf{s} = g :: to :: io :: is :: oo :: os :: s \\ to_a = to \bmod 2^{160} \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(0, 1, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(0, 1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu' = \mu[\mathsf{i} \to aw][\mathsf{s} \to 0 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c][\mathsf{m} \to \mu.\mathsf{m}[[oo, oo + s - 1] \to d]][\mathsf{d}_r \to d]\end{array}}{\Gamma \vDash REVERT(gas, d) :: (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CREATE} \\ \mu.\mathsf{s} = va :: io :: is :: s \qquad aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 32000 \\ \mu' = \mu[\mathsf{s} \to 0 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c][\mathsf{i} \to aw][\mathsf{d}_r \to d]\end{array}}{\Gamma \vDash REVERT(gas, d) :: (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma, \eta) :: S}$$

Note that this case is under-specified in the yellow paper. However, the improvement proposal states that in the case of contract creation, the REVERT should not result in any change of the

global storage, and the data should be made available in the return data buffer (for communicating error codes). Note that this is different from returning from a contract creation with a halting state where the data is interpreted as code and not made available in the return data buffer.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{STATICCALL} \qquad \mu.\mathsf{s} = g :: to :: io :: is :: oo :: os :: s \\ to_a = to \mod 2^{160} \qquad \mathit{flag} = \sigma.to_a = \bot \,?\, 0 : 1 \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(0, \mathit{flag}, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(0, \mathit{flag}) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu' = \mu[\mathsf{i} \to aw][\mathsf{s} \to 0 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c][\mathsf{m} \to \mu.\mathsf{m}[[oo, oo + s - 1] \to d]][\mathsf{d}_r \to d]\end{array}}{\Gamma \vDash \mathit{REVERT}(gas, d) :: (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma, \eta) :: S}$$

EXTCODEHASH The EXTCODEHASH opcode [JB18] (introduced in the Constantinople hard fork) allows for accessing the Keccak hash of an external contract code. So far it was possible to access the size of an external contract using EXTCODESIZE and its code using EXTCODECOPY what would allow for computing the hash of another contract's code. However, this would be very expensive. For this reason it was decided to provide native support for this functionality.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{EXTCODEHASH} \qquad \mu.\mathsf{s} = a :: s \qquad \mathit{code} = \sigma(a \mod 2^{160}).\mathsf{code} \\ \mathit{valid}\,(\mu.\mathsf{gas}, 400, |\mu.\mathsf{s}|) \qquad \sigma(a \bmod 2^{160}) \neq \bot \qquad h = \mathit{Keccak}(\mathit{code}) \\ \mu' = \mu[\mathsf{s} \to h :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 400] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{EXTCODEHASH}_c(a)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{EXTCODEHASH} \qquad \mu.\mathsf{s} = a :: s \qquad \mathit{code} = \sigma(a \mod 2^{160}).\mathsf{code} \\ \mathit{valid}\,(\mu.\mathsf{gas}, 400, |\mu.\mathsf{s}|) \qquad (\sigma(a \bmod 2^{160}) = \bot \vee \mathit{isEmpty}(\sigma(a \bmod 2^{160}))) \\ \mu' = \mu[\mathsf{s} \to 0 :: \mu.\mathsf{s}][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 400] \qquad c = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code})\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{EXTCODEHASH}_c(a)} (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \mathsf{EXTCODEHASH} \qquad (\neg \mathit{valid}\,(\mu.\mathsf{gas}, 400, |\mu.\mathsf{s}|) \vee |\mu.\mathsf{s}| < 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to \mathit{EXC} :: S}$$

Bitwise Shifting Instructions The instruction set was in the Constantinople hard fork extended with binary operations for bitwise shifting [BB17]. These include arithmetic left shifts SHL, arithmetic right shifts SHR, and logical right shifts SAR. Since those are simple binary operations, their rules are analogous to the ones of other binary operations, and we added them to the general rule for binary operations presented in the previous section.

CREATE2 The new opcode CREATE2 (also called *skinny CREATE*) [But18] was introduced as part of the Constantinople hard fork to allow for the creation of contracts at predetermined addresses. It is similar to the CREATE opcode with the main difference that the address of the new contract does not depend on the actor's nonce but instead on the initialization code and a user-defined salt. Another change affects the gas cost, which now consider the cost for hashing the initialization code in the course of the address computation.

We give the small-step rules for this opcode:

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CREATE2} \qquad \iota.\mathsf{f_{mod}} = 1 \qquad \mu.\mathsf{s} = va :: io :: is :: salt :: s \\ aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 32000 + 6 * \lceil \frac{is}{32} \rceil \\ valid\,(\mu.\mathsf{gas}, c, |s|+1) \qquad va \leq \sigma(\iota.\mathsf{actor}).\mathsf{balance} \qquad |S|+1 \leq 1024 \\ \rho = newAddressSkinny(\iota.\mathsf{actor}, salt, i) \qquad \sigma(\rho) \neq \bot \qquad b = \sigma(\rho).\mathsf{balance} + va \\ \sigma' = \sigma\langle \rho \to (1, b, \lambda x.\,0, \epsilon)\rangle\langle \iota.\mathsf{actor} \to \sigma(\iota.\mathsf{actor})[\mathsf{balance} \mathrel{-}= va][\mathsf{nonce} \mathrel{+}= 1]\rangle \\ i = \mu.\mathsf{m}\,[io, io + is - 1] \\ \iota' = \iota[\mathsf{sender} \to \iota.\mathsf{actor}][\mathsf{actor} \to \rho][\mathsf{value} \to va][\mathsf{code} \to i][\mathsf{input} \to \epsilon] \\ \mu' = (L\,(\mu.\mathsf{gas} - c), 0, \lambda x.\,0, 0, \epsilon, \epsilon) \\ \eta' = \eta[\mathsf{S}_{\not\iota} \to \eta.\mathsf{S}_{\not\iota} \cup \{\iota.\mathsf{actor}, \rho\}] \qquad c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code}) \end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{CREATE2}_{c'}(va,io,is,salt)} (\mu', \iota', \sigma', \eta') :: (\mu, \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CREATE2} \qquad \iota.\mathsf{f_{mod}} = 1 \qquad \mu.\mathsf{s} = va :: io :: is :: salt :: s \\ aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 32000 + 6 * \lceil \frac{is}{32} \rceil \qquad valid\,(\mu.\mathsf{gas}, c, |s|+1) \\ va \leq \sigma(\iota.\mathsf{actor}).\mathsf{balance} \qquad |S|+1 \leq 1024 \qquad \rho = newAddressSkinny(\iota.\mathsf{actor}, salt, i) \\ \sigma(\rho) \neq \bot \qquad \sigma(\rho).\mathsf{nonce} = 0 \qquad \sigma(\rho).\mathsf{code} = \epsilon \qquad b = \sigma(\rho).\mathsf{balance} + va \\ \sigma' = \sigma\langle \rho \to (1, b, \lambda x.\,0, \epsilon)\rangle\langle \iota.\mathsf{actor} \to \sigma(\iota.\mathsf{actor})[\mathsf{balance} \mathrel{-}= va][\mathsf{nonce} \mathrel{+}= 1]\rangle \\ i = \mu.\mathsf{m}\,[io, io + is - 1] \\ \iota' = \iota[\mathsf{sender} \to \iota.\mathsf{actor}][\mathsf{actor} \to \rho][\mathsf{value} \to va][\mathsf{code} \to i][\mathsf{input} \to \epsilon] \\ \mu' = (L\,(\mu.\mathsf{gas} - c), 0, \lambda x.\,0, 0, \epsilon, \epsilon) \\ \eta' = \eta[\mathsf{S}_{\not\iota} \to \eta.\mathsf{S}_{\not\iota} \cup \{\iota.\mathsf{actor}, \rho\}] \qquad c' = (\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{code}) \end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \xrightarrow{\mathsf{CREATE2}_{c'}(va,io,is,salt)} (\mu', \iota', \sigma', \eta') :: (\mu, \iota, \sigma, \eta) :: S}$$

Where $newAddressSkinny \in \mathcal{A} \times \mathbb{N}_{256} \times [\mathbb{B}^8] \to \mathcal{A}$ is defined as follows:

$$newAddressSkinny(addr, salt, d) = Keccak(255 concat addr \cdot salt \cdot Keccak(d))[96, 255]$$

The following rules describe failure at call time:

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CREATE2} \qquad \iota.\mathsf{f_{mod}} = 1 \qquad \mu.\mathsf{s} = va :: io :: is :: s \\ i = \mu.\mathsf{m}\,[io, io + is - 1] \qquad \rho = newAddressSkinny(\iota.\mathsf{actor}, salt, i) \\ \sigma(\rho) \neq \bot \qquad (\sigma(\rho).\mathsf{nonce} > 0 \vee \sigma(\rho).\mathsf{code} \neq \epsilon) \end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to EXC :: (\mu, \iota, \sigma, \eta) :: S}$$

$$\frac{\omega_{\mu,\iota} = \text{CREATE2} \qquad \iota.\mathsf{f}_{\mathsf{mod}} = 1 \qquad \mu.\mathsf{s} = va :: io :: is :: salt :: s \qquad aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 32000 + 6 * \lceil \frac{is}{32} \rceil \qquad valid\,(\mu.\mathsf{gas}, c, |s| + 1) \qquad (va > \sigma(\iota.\mathsf{actor}).\mathsf{balance} \vee |S| + 1 > 1024)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow EXC :: (\mu, \iota, \sigma, \eta) :: S}$$

In addition the usual out-of-gas exception and violations of the stack limits need to be considered:

$$\frac{\omega_{\mu,\iota} = \text{CREATE2} \qquad \mu.\mathsf{s} = va :: io :: is :: salt :: s \qquad aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 32000 + 6 * \lceil \frac{is}{32} \rceil \qquad \neg valid\,(\mu.\mathsf{gas}, c, |s| + 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow EXC :: S}$$

$$\frac{\omega_{\mu,\iota} = \text{CREATE2} \qquad (\mu.\mathsf{s} < 4 \vee \iota.\mathsf{f}_{\mathsf{mod}} = 1)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow EXC :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \text{CREATE2} \qquad \mu.\mathsf{s} = va :: io :: is :: salt :: s \\ aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 320006 * \lceil \frac{is}{32} \rceil \\ |d| \leq 24576 \qquad c_{final} = 200 \cdot |d| \qquad gas \geq c_{final} \qquad \rho = newAddressSkinny(\iota.\mathsf{actor}, salt, i) \\ \mu' = \mu[\mathsf{s} \rightarrow \rho :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c - c_{final}][\mathsf{i} \rightarrow aw][\mathsf{d}_r \rightarrow \epsilon] \\ \sigma'' = \sigma'\langle \rho \rightarrow \sigma'(\rho)[\mathsf{code} \rightarrow d]\rangle\end{array}}{\Gamma \vDash HALT(\sigma', gas, d, \eta') :: (\mu, \iota, \sigma, \eta) :: S \;\rightarrow (\mu', \iota, \sigma'', \eta') :: S}$$

$$\frac{\omega_{\mu,\iota} = \text{CREATE2} \qquad c_{final} = 200 \cdot |d| \qquad (gas < c_{final} \vee |d| > 24576)}{\Gamma \vDash HALT(\sigma', gas, d, \eta') :: (\mu, \iota, \sigma, \eta) :: S \;\rightarrow EXC :: (\mu, \iota, \sigma, \eta) :: S :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \text{CREATE2} \qquad \mu.\mathsf{s} = va :: io :: is :: salt :: s \qquad aw = M\,(\mu.\mathsf{i}, io, is) \\ c = C_{mem}\,(\mu.\mathsf{i}, aw) + 32000 + 6 * \lceil \frac{is}{32} \rceil \qquad c_{final} = 200 \cdot |d| \qquad |d| \leq 24576 \\ gas \geq c_{final} \qquad \rho = newAddressSkinny(\iota.\mathsf{actor}, salt, i) \qquad (\sigma'(\rho) = \bot \;\vee\; isEmpty(\sigma'(\rho))) \\ \mu' = \mu[\mathsf{s} \rightarrow \rho :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c - c_{final}][\mathsf{i} \rightarrow aw][\mathsf{d}_r \rightarrow \epsilon] \\ \sigma'' = \sigma'\langle \rho \rightarrow \bot\rangle\end{array}}{\Gamma \vDash HALT(\sigma', gas, d, \eta') :: (\mu, \iota, \sigma, \eta) :: S \;\rightarrow (\mu', \iota, \sigma'', \eta') :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CREATE2} \qquad \mu.\mathsf{s} = va :: io :: is :: salt :: s \\ aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 32000 + 6 * \lceil \frac{is}{32} \rceil \\ \mu' = \mu[\mathsf{s} \to 0 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c][\mathsf{i} \to aw][\mathsf{d}_r \to d]\end{array}}{\Gamma \vDash \mathit{REVERT}(gas, d) :: (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CREATE2} \qquad \mu.\mathsf{s} = va :: io :: is :: salt :: s \\ aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 32000 + 6 * \lceil \frac{is}{32} \rceil \\ \mu' = \mu[\mathsf{s} \to 0 :: s][\mathsf{pc} \mathrel{-}= 1][\mathsf{gas} \mathrel{+}= c][\mathsf{i} \to aw][\mathsf{d}_r \to \epsilon]\end{array}}{\Gamma \vDash \mathit{EXC} :: (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma, \eta) :: S}$$

Note that additionally to the discussed changes, the concrete gas costs of the individual opcodes are frequently changing. We do not overview these changes here since the concrete gas costs do not impact the theoretical considerations discussed in this thesis.

A.4 Auxiliary Definitions

For defining the security notions, we consider parts of the traces produced by the execution. To this end, we use projection functions that filter only specific actions of a trace that we define formally in the following:

Definition 12 (Projection on Execution Traces)**.** *Let $f \in Act \to \mathbb{B}$ be a filtering function. Then the projection on traces is recursively defined as follows*

$$\pi \downarrow_f = \begin{cases} \epsilon & \pi = \epsilon \\ a :: (\pi' \downarrow_f) & \pi = a :: \pi' \wedge f(a) = 1 \\ \pi' \downarrow_f & \pi = a :: \pi' \wedge f(a) = 0 \end{cases}$$

For projecting on the call events of a trace, we define calls_c, the function filtering all call and create actions of contract c:

$$\mathsf{calls}_c(a) = \begin{cases} 1 & a = \mathsf{CALL}_c(g, to, va, io, is, oo, os) \\ & \vee\, a = \mathsf{CREATE}_c(va, io, is) \\ & \vee\, a = \mathsf{CREATE2}_c(va, io, is, salt) \\ & \vee\, a = \mathsf{CALLCODE}_c(g, to, va, io, is, oo, os) \\ & \vee\, a = \mathsf{DELEGATECALL}_c(g, to, io, is, oo, os) \\ & \vee\, a = \mathsf{STATICCALL}_c(g, to, io, is, oo, os) \\ & \text{for some } g, to, va, io, is, oo, os, salt \in \mathbb{B}^{256} \\ 0 & \text{otherwise} \end{cases}$$

Throughout the definitions, we use the notion of concatenation $(++)$ for (plain annotated) call stacks.

Definition 13 (Concatenation of Call Stacks)**.**

$$++ \in S \times S \to \mathbb{S}$$
$$S ++ U := \begin{cases} U & S = \epsilon \\ s :: (S' ++ U) & S = s :: S' \end{cases}$$

Note that we will usually not make a distinction between plain call stacks and call stacks. Also, when writing $s :: S$ we will assume s to potentially refer to a final execution state.

We will define the predicates $\mathit{final}\,(\cdot)$, $\mathit{isRegular}(\cdot)$ and $\mathit{isHalt}(\cdot)$ to determine whether the corresponding execution state is a final, a regular or a halting execution state, respectively.

$$\mathit{final}\,(\cdot) \in \mathcal{S}_{+ter} \to \mathbb{B}$$
$$\mathit{final}\,(s) := \begin{cases} 1 & s = \mathit{EXC} \ \vee \ \exists \sigma \in \Sigma, d \in [\mathbb{B}^8], g \in \mathbb{N}, \eta \in N.\ s = \mathit{HALT}(\sigma, g, d, \eta) \\ & \quad \vee \exists d \in [\mathbb{B}^8], g \in \mathbb{N}.\ s = \mathit{REVERT}(g, d) \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{isRegular}(\cdot) \in \mathcal{S}_{+ter} \to \mathbb{B}$$
$$\mathit{isRegular}(s) := 1 - \mathit{final}\,(s)$$

$$\mathit{isHalt}(\cdot) \in \mathcal{S}_{+ter} \to \mathbb{B}$$
$$\mathit{isHalt}(s) := \begin{cases} 1 & \exists \sigma \in \Sigma, d \in [\mathbb{B}^8], g \in \mathbb{N}, \eta \in N.\ s = \mathit{HALT}(\sigma, g, d, \eta) \\ 0 & \text{otherwise} \end{cases}$$

Where

$$\begin{aligned} \mathcal{S}_{+ter} = \mathcal{S} \ &\cup \ \{\mathit{EXC}\} \\ &\cup \ \{\mathit{HALT}(\sigma, g, d, \eta) \mid \sigma \in \Sigma \ \wedge \ d \in [\mathbb{B}^8] \ \wedge \ g \in \mathbb{N} \ \wedge \ \eta \in N\} \\ &\cup \ \{\mathit{REVERT}(g, d) \mid d \in [\mathbb{B}^8] \ \wedge \ g \in \mathbb{N}\} \end{aligned}$$

A.5 Transaction Execution

In the following, we formally define transaction execution.

Formally, a transaction is a tuple $(\mathit{nonce}, \mathit{price}, \mathit{gaslimit}, \mathit{to}, \mathit{value}, \mathit{sender}, \mathit{input}, \mathit{sig})$ where

- *nonce* $\in \mathbb{N}_{256}$ is a number counting the number of transactions issued by the sender
- *price* $\in \mathbb{N}_{256}$ is the amount of *wei* to pay for one unit of gas when executing this transaction
- *gaslimit* $\in \mathbb{N}_{256}$ is the maximum amount of gas to be spent on the execution of the transaction
- *to* $\in \mathbb{N}_{160} \cup \{\bot\}$ is the recipient of the the transaction. If the recipient is $\bot$ then this indicates a contract creating transaction.
- *value* $\in \mathbb{N}_{256}$ is the amount of *wei* transferred by the transaction
- *sender* $\in \mathbb{N}_{160}$ is the sender of the transaction
- *input* $\in [\mathbb{B}^8]$ is the input given to the transaction. This might either be the arguments given to a contract in case of a call transaction or the byte code that initializes the newly created contract in the case of a create transaction
- *sig* is the signature of the transaction

Note that we here simplified some components, e.g., we assume the sender to be an explicit field of the transaction while in the concrete Ethereum implementation, the sender address would be recovered from the signature information. Accordingly we treat the signature of the transaction in an abstract fashion, simply assuming that there is a function checkSig such that $\mathsf{checkSig}(\mathit{sig}, T, \mathit{sender}) = 1$ if and only Sig is a valid transaction of address (public key) *sender* for the transaction T.

Technically for determining the validity of a transaction, it is not sufficient to consider one transaction in isolation, but the previously executed transactions in a block need to be considered as well. This is since a block (similarly to the individual transaction) specifies a gas limit that may not be exceeded. For this reason, it is required to slightly update the initialization function to take the remaining block gas budget into account. Further, the finalization function should return the gas returned from the execution. For the sake of completeness, we also make the finalization function return the logs of transaction execution (even though those do not influence the global blockchain state).

$$\begin{aligned} &\mathit{initalizeGas}(\cdot) \in \mathcal{T} \to \mathbb{N}_{256} \\ &\mathit{initalizeGas}(T) = \left(\sum_{b \in T.\mathsf{input}} 1_{\{0\}}(b) * 4 + 1_{\mathbb{N}_8/\{0\}}(b) * 68 \right) + 1_{\{\bot\}}(T.\mathsf{to}) * 32000 + 21000 \end{aligned}$$

$$\begin{aligned} \mathit{isValidT}(\cdot,\cdot,\cdot,\cdot) &\in \mathcal{T} \times \mathcal{H} \times \Sigma \times \mathbb{N}_{256} \to \mathbb{B} \\ \mathit{isValidT}(T, H, \sigma, g) &= \mathsf{checkSig}(T.\mathsf{sig}, T, \mathit{sender}) = 1 \\ &\quad \wedge\ \sigma(T.\mathsf{sender}) \neq \bot\ \wedge\ T.\mathsf{nonce} = \sigma(T.\mathsf{sender}).\mathsf{n} \\ &\quad \wedge\ \mathit{initalizeGas}(T) \leq T.\mathsf{gaslimit} \\ &\quad \wedge\ T.\mathsf{gaslimit} * T.\mathsf{price} + T.\mathsf{value} \leq \sigma(T.\mathit{sender}).\mathsf{b} \\ &\quad \wedge\ T.\mathsf{gaslimit} \leq g \end{aligned}$$

$$\mathit{initializeT}(\cdot,\cdot,\cdot,\cdot) \in \mathcal{T} \times \mathcal{H} \times \Sigma \times \mathbb{N}_{256} \to (\mathcal{T}_{\mathit{env}} \times \mathcal{S}_{\mathit{annotated}}) \cup \{\bot\}$$

$$\mathit{initializeT}(T,H,\sigma,g) = \begin{cases} (\Gamma, s_c) & \mathit{isValidT}(T,H,\sigma,g) \\ & \wedge\, T.\mathsf{to} \neq \bot \\ & \wedge\, \Gamma = (T.\mathsf{sender}, T.\mathsf{price}, H) \\ & \wedge\, s.\mu = (T.\mathsf{gaslimit} - \mathit{initalizeGas}(T), 0, \lambda x.\, 0, 0, \epsilon, \epsilon) \\ & \wedge\, s.\iota = (T.\mathsf{to}, T.\mathsf{input}, T.\mathsf{sender}, T.\mathsf{value}, \sigma(T.\mathsf{to}).\mathit{code}, 1) \\ & \wedge\, s.\sigma = \sigma\langle T.\mathsf{sender} \to \sigma(T.\mathsf{sender})[\mathsf{b} \mathrel{-}= T.\mathsf{gaslimit} * T.\mathit{price}][\mathsf{n} \mathrel{+}= 1]\rangle \\ & \wedge\, s.\eta = (0, \epsilon, \emptyset, \{T.\mathsf{sender}, T.\mathsf{to}, H.\mathsf{beneficiary}\}) \\ & \wedge c = (T.\mathsf{to}, \sigma(T.\mathsf{to}).\mathsf{code}) \\ (\Gamma, s_c) & \mathit{isValidT}(T,H,\sigma,g) \\ & \wedge\, T.\mathsf{to} = \bot \\ & \wedge\, \Gamma = (T.\mathsf{sender}, T.\mathsf{price}, H) \\ & \wedge\, s.\mu = (T.\mathsf{gaslimit} - \mathit{initalizeGas}(T), 0, \lambda x.\, 0, 0, \epsilon, \epsilon) \\ & \wedge \rho = \mathit{newAddress}\,(T.\mathit{sender}, T.\mathsf{nonce}) \\ & \wedge c = (\rho, \epsilon) \\ & \wedge\, s.\iota = (\rho, \epsilon, T.\mathsf{sender}, T.\mathsf{value}, T.\mathit{input}, 1) \\ & \wedge\, s.\sigma = \sigma\langle T.\mathsf{sender} \to \sigma(T.\mathsf{sender})[\mathsf{b} \mathrel{-}= T.\mathsf{gaslimit} * T.\mathsf{price}][\mathsf{n} \mathrel{+}= 1]\rangle \\ & \wedge\, s.\eta = (0, \epsilon, \emptyset, \{T.\mathsf{sender}, \rho, H.\mathsf{beneficiary}\}) \\ \bot & \text{otherwise} \end{cases}$$

One should note that the initialization depends on whether the transaction corresponds to a contract creation or a contract call. If the recipient ($T.\mathsf{to}$) of the transaction is $\bot$, then the transaction is interpreted as contract creation and the input $T.\mathsf{input}$ to the transaction is interpreted as contract creation code. Otherwise, the input is considered to be the input to the contract call. When initializing the transaction first, its validity is checked. This is done by the function *isValidT*. For the validity, it is checked that the transaction has a valid signature, that the sender of the transaction exists and that the nonce of its account state corresponds to the transaction nonce. Further, it is checked that the gas needed to initialize the transaction (as calculated by *initalizeGas*) does not exceed the transaction gas limit and that the sender has a sufficient balance to pay for the gas potentially needed for the transaction (according to the gas limit). Finally, it is checked that the allocated gas limit does not exceed the remaining gas that might be consumed by the block (here given as extra argument g).

The initialization itself then closely resembles the initiation of internal call or create abstractions as triggered by CALL, and CREATE instructions, respectively.

Next we formally define how a transaction is finalized based on the final execution state and the transaction effects:

$$\mathit{getGas}(\cdot) \in \mathcal{S} \to \mathbb{N}_{256}$$

$$\mathit{getGas}(s) = \begin{cases} \mu.\mathsf{gas} & s = (\mu, \iota, \sigma, \eta) \\ g & \mathit{HALT}(\sigma, g, d, \eta) \\ 0 & \mathit{EXC} \end{cases}$$

$$\mathit{finalizeGas}(\cdot,\cdot,\cdot) \in \mathcal{S} \times N \times \mathcal{T} \to \mathbb{N}_{256}$$
$$\mathit{finalizeGas}(s,\eta,T) = \mathit{getGas}(s) + \mathit{min}\left(\left\lfloor \frac{T.\mathsf{gaslimit} - \mathit{getGas}(s)}{2} \right\rfloor, \eta.\mathsf{balance} + |\eta.\mathsf{S}_\dagger| * 24000\right)$$

$$\begin{aligned}
&\mathit{finalizeT}(\cdot,\cdot,\cdot) \in \mathcal{S} \times N \times \mathcal{T} \to \Sigma \times \mathbb{N}_{256} \times L \\
&\mathit{finalizeT}(s,\eta,T) = (\sigma\langle T.\mathsf{sender} \to \sigma(T.\mathsf{sender})[\mathsf{b} \mathrel{+}= \mathit{finalizeGas}(s,\eta,T) * T.\mathsf{price}]\rangle \\
&\qquad\qquad \langle H.\mathsf{beneficiary} \to \sigma(H.\mathsf{beneficiary})[\mathsf{b} \mathrel{+}= (T.\mathsf{gaslimit} - \mathit{finalizeGas}(s,\eta,T) * T.\mathsf{price})]\rangle \\
&\qquad\qquad \langle a \to \bot\rangle_{a \in \eta.\mathsf{S}_\dagger \vee a \in S_\xi \wedge \mathit{isEmpty}(a)}, \\
&\qquad T.\mathsf{gaslimit} - \mathit{finalizeGas}(s,\eta,T), \\
&\qquad \eta.\mathsf{L})
\end{aligned}$$

The function *finalizeGas* calculates the remaining gas after the execution. To this end, it does not only consider the gas left in the final execution state s but also considers the refund balance collected in the transaction effects. These components, however, are only refunded up to a cap of half the difference between the remaining gas and the gas limit. Finally, the sender gets refunded according to the remaining gas as calculated by the *finalizeGas* function, and the beneficiary of the transaction receives the gas that is left after that from the prepaid fees that the sender paid prior to the execution. Finally, all contracts recorded in the suicide set $\eta.\mathsf{S}_\dagger$ get deleted from the global state. The *finalizeT* function additionally returns the gas that was paid to the beneficiary (since this counts towards the block gas limit), and for the sake of completeness, the logs $\eta.\mathsf{L}$ produced by the transaction.

In the following, we will use the defined functions to characterize the execution of a whole block. We will assume the slightly simplified presentation of a block to be a pair of a block header and a list of transactions.

$$\frac{}{\sigma \xrightarrow{\epsilon,H} (\sigma, H.\mathsf{gaslimit}, \epsilon)}$$

$$\frac{\sigma \xrightarrow{ts,H} (\sigma', g', ls) \qquad (\Gamma, s_c) = \mathit{initializeT}(T,H,\sigma,g') \qquad \Gamma \vDash s_c :: \epsilon \to^* s'_c :: \epsilon \qquad \mathit{final}\,(s') \qquad (\sigma'', g'', L) = \mathit{finalizeT}(s',\eta',T)}{\sigma \xrightarrow{ts\cdot[T],H} (\sigma'', g' - g'', ls \cdot [L])}$$

Note that the remaining gas g' from the execution of the prior transactions in the block is used to check the validity of the transaction in the *initializeT* function. This remaining gas is then decreased according to the gas usage of the transaction.

A.6 Properties of the Small-step Semantics

We first state some general properties of the small-step semantics.

Lemma 1 (Determinism). *Let* $\Gamma \vDash S \xrightarrow{a} S'$ *and* $\Gamma \vDash S \xrightarrow{b} S''$ *then* $S' = S''$ *and* $a = b$.

Proof. By an exhaustive case analysis on the small-step rules. The small step relation is functional (w.r.t. to the transaction environment and call stack). □

Lemma 2 (Progress). *Let* Γ *be a transaction environment and* S *be a call stack. Then either* $S = [s]$ *and final* (s) *or there is some* S' *and* a *such that* $\Gamma \vDash S \xrightarrow{a} S'$.

Proof. By an exhaustive case analysis of the small step rules. □

The progress lemma implies that execution cannot get stuck.

We can also show the following local progress property:

Lemma 3 (Local Progress). *Let* Γ *be a transaction environment and* S *be a call stack and* s *be an execution state. Then either final* (s) *or there is some* S' *with* $1 \leq |S| \leq 2$, *and* a *such that* $\Gamma \vDash s :: S \xrightarrow{a} S' +\!\!+ S$.

Proof. By an exhaustive case analysis of the small step rules. □

The following lemmas summarize some general properties of call stack evolution during the execution. The small-step semantics is designed such that the call stack records the execution state as at the point of calling. The corresponding states only get modified when returning from an internal transaction. In this case, modification is guaranteed, since the gas for the execution is subtracted. As a consequence, an unmodified (sub) call stack indicates that the execution of the same internal transaction is still executed.

To reason about this behavior, we establish an order on call stacks that we will show to decrease during execution.

Definition 14 (Call Stack Precedence). *Let* S_1, S_2 *be (annotated) call stacks. We say that* S_1 *precedes* S_2 *(written* $S_1 \prec S_2$ *iff the following condition holds:*

$$\begin{aligned}\exists\, S\ S_1'\ S_2'.\ & |S_1'| > 0 \\ \wedge\ & S_1 = S_1' +\!\!+ S \\ \wedge\ & S_2 = S_2' +\!\!+ S \\ \wedge\ & (|S_2'| > 0 \Rightarrow \mathit{getGas}^-(\mathit{last}(S_1')) < \mathit{getGas}^-(\mathit{last}(S_2')))\end{aligned}$$

Recall that $\mathit{last}(S)$ denotes the last element of a call stack (yielding $\bot$ in case of an empty call stack). Here getGas^- denotes the following slightly modified version of the previously introduced function getGas:

$$\mathit{getGas}^-(\cdot) \in \mathcal{S} \to \mathbb{N}_{256}$$

$$\mathit{getGas}^-(s) = \begin{cases} \mu.\mathsf{gas} & s = (\mu, \iota, \sigma, \eta) \\ g & \mathit{HALT}(\sigma, g, d, \eta) \\ -1 & \mathit{EXC} \end{cases}$$

Note that the decision to define the gas value of *EXC* to be -1 will facilitate later reasoning since it ensures that the gas in local execution always strictly decreases since regular execution states can have a gas value of 0 and then only run out of gas in the next step.

We show that $\prec$ is a strict partial order on call stacks.

Lemma 4. *$\prec$ is a strict partial order on (annotated) call stacks.*

Proof. We show that $\prec$ is irreflexive, transitive, and asymmetric.

- *Irreflexivity.* Let S be an (annotated) call stack. We show that $S \not\prec S$. Towards contradiction we assume that $S \prec S$. Then by Definition 14, there exists S', S_1, S_2 s.t. (1) $|S_1| > 0$, (2) $S = S_1 +\!\!+ S'$, (3) $S = S_2 +\!\!+ S'$, and (4) $|S_2| > 0 \Rightarrow \mathit{getGas}^-(\mathit{last}(S_1)) < \mathit{getGas}^-(\mathit{last}(S_2))$. By (2) and (3) we know that $S_1 +\!\!+ S' = S_2 +\!\!+ S'$ and hence $S_1 = S_2$. Since $|S_1| > 0$ (1) we can hence conclude by (4) that $\mathit{getGas}^-(\mathit{last}(S_1)) < \mathit{getGas}^-(\mathit{last}(S_1))$ which is a clear contradiction.

- *Transitivity.* Let S_1, S_2, S_3 be (annotated) call stacks. We show that $S_1 \prec S_2 \wedge S_2 \prec S_3 \Rightarrow S_1 \prec S_3$. Assume that (A) $S_1 \prec S_2$ and (B) $S_2 \prec S_3$. Then by Definition 14, there exist S_A, S_1', S_2' s.t. (A1) $|S_1'| > 0$, (A2) $S_1 = S_1' +\!\!+ S_A$, (A3) $S_2 = S_2' +\!\!+ S_A$, and (A4) $|S_2'| > 0 \Rightarrow \mathit{getGas}^-(\mathit{last}(S_1')) < \mathit{getGas}^-(\mathit{last}(S_2'))$ and further there exist S_B S_2'', S_3' s.t. (B1) $|S_2''| > 0$, (B2) $S_2 = S_2'' +\!\!+ S_B$, (B3) $S_3 = S_3' +\!\!+ S_B$, and (B4) $|S_3'| > 0 \Rightarrow \mathit{getGas}^-(\mathit{last}(S_2'')) < \mathit{getGas}^-(\mathit{last}(S_3'))$. Then by (A3) and (B2) we get that (C1) $S_2' +\!\!+ S_A = S_2 = S_2'' +\!\!+ S_B$. So then either $S_A = U +\!\!+ S_B$ for some U or $S_B = U +\!\!+ S_A$ for some U. We make a case distinction.

  1. Assume that $S_A = U +\!\!+ S_B$ for some U with $|U| > 0$. Then $S_1 = (S_1' +\!\!+ U) +\!\!+ S_B$ and $S_3 = S_3' +\!\!+ S_B$. Now assume that $|S_3'| > 0$. We show that $\mathit{getGas}^-(\mathit{last}(S_1' +\!\!+ U)) < \mathit{getGas}^-(\mathit{last}(S_3'))$. Since $|U| > 0$ it is sufficient to show that $\mathit{getGas}^-(\mathit{last}(U)) < \mathit{getGas}^-(\mathit{last}(S_3'))$. By (B4) we already know that (C2) $\mathit{getGas}^-(\mathit{last}(S_2'')) < \mathit{getGas}^-(\mathit{last}(S_3'))$. From (C1) we can conclude that $S_2' +\!\!+ U +\!\!+ S_B = S_2'' +\!\!+ S_B$ and hence $S_2'' = S_2' +\!\!+ U$. So consequently by (C2) $\mathit{getGas}^-(\mathit{last}(S_2' +\!\!+ U)) < \mathit{getGas}^-(\mathit{last}(S_3'))$ and since $|U| > 0$ also $\mathit{getGas}^-(\mathit{last}(U)) < \mathit{getGas}^-(\mathit{last}(S_3'))$ what closes the case.
  2. Assume that $S_B = U +\!\!+ S_A$ for some U. Then $S_1 = S_1' +\!\!+ S_A$ and $S_3 = S_3' +\!\!+ U +\!\!+ S_A$. Now assume that $|S_3'| > 0$. We show that $\mathit{getGas}^-(\mathit{last}(S_1')) < \mathit{getGas}^-(\mathit{last}(S_3' +\!\!+ U))$. By (B4) we already know that (C2) $\mathit{getGas}^-(\mathit{last}(S_2'')) < \mathit{getGas}^-(\mathit{last}(S_3'))$. From (C1) we can conclude that $S_2' +\!\!+ S_A = S_2'' +\!\!+ U +\!\!+ S_A$ and hence $S_2' = S_2'' +\!\!+ U$. Since $|S_2''| > 0$ (B1) we know that also $|S_2'| > 0$ and hence by (A4) we can conclude that $\mathit{getGas}^-(\mathit{last}(S_1')) < \mathit{getGas}^-(\mathit{last}(S_2'))$ and consequently (C3) $\mathit{getGas}^-(\mathit{last}(S_1')) < \mathit{getGas}^-(\mathit{last}(S_2'' +\!\!+ U))$. We do a case distinction on $|U|$.
     - if $U = \epsilon$ then the claim trivially holds by (C2) and (C3).
     - if $|U| > 0$ then from (C3) we get that $\mathit{getGas}^-(\mathit{last}(S_1')) < \mathit{getGas}^-(\mathit{last}(U))$ and at the same time the claim to be shown also reduces to $\mathit{getGas}^-(\mathit{last}(S_1')) < \mathit{getGas}^-(\mathit{last}(U))$ what concludes the case.

- *Asymmetry.* Let S_1, S_2 be (annotated) call stacks. We show that $S_1 \prec S_2 \Rightarrow S_2 \not\prec S_1$. Assume that $S_1 \prec S_2$. Then by Definition 14, there exists S, S'_1, S'_2 s.t. (A1) $|S'_1| > 0$, (A2) $S_1 = S'_1 +\!\!+ S$, (A3) $S_2 = S'_2 +\!\!+ S$, and (A4) $|S'_2| > 0 \Rightarrow \mathit{getGas}^-(\mathit{last}(S'_1)) < \mathit{getGas}^-(\mathit{last}(S'_2))$. Further assume towards contradiction that $S_2 \prec S_1$. Then by Definition 14, there exists S', S''_1, S''_2 s.t. (B1) $|S''_2| > 0$, (B3) $S_1 = S''_1 +\!\!+ S'$, (B2) $S_2 = S''_2 +\!\!+ S'$, and (B4) $|S''_1| > 0 \Rightarrow \mathit{getGas}^-(\mathit{last}(S''_2)) < \mathit{getGas}^-(\mathit{last}(S''_1))$. By (A2) and (B3) we know that (C1) $S'_1 +\!\!+ S = S''_1 +\!\!+ S'$. So we know that either $S' = U +\!\!+ S$ for some U or $S = U +\!\!+ S'$ for some U with $|U| > 0$. We distinguish the two cases.

  1. Assume that $S' = U +\!\!+ S$ for some U. Then by (C1) we know that (C2) $S'_1 +\!\!+ S = S''_1 +\!\!+ U +\!\!+ S$ and hence $S'_1 = S''_1 +\!\!+ U$. By (A3) and (B2) we know that $S'_2 +\!\!+ S = S''_2 +\!\!+ S'$ and consequently $S'_2 +\!\!+ S = S''_2 +\!\!+ U +\!\!+ S$ and so $S'_2 = S''_2 +\!\!+ U$. We make a case distinction on U.
     - Assume that $U = \epsilon$. Then $S'_1 = S''_1$ and $S'_2 = S''_2$. Since $|S'_1| > 0$ (A1) we get from (B4) that (C3) $\mathit{getGas}^-(\mathit{last}(S'_2)) < \mathit{getGas}^-(\mathit{last}(S'_1)$. Similarly, since $|S''_2| > 0$ (B1) we get from (A4) that $\mathit{getGas}^-(\mathit{last}(S'_1)) < \mathit{getGas}^-(\mathit{last}(S'_2)$ which clearly contradicts (C3).
     - Assume that $|U| > 0$. Then clearly $|S'_2| > 0$ and by (A4) $\mathit{getGas}^-(\mathit{last}(S'_2)) < \mathit{getGas}^-(\mathit{last}(S'_1)$, and so $\mathit{getGas}^-(\mathit{last}(S''_2 +\!\!+ U)) < \mathit{getGas}^-(\mathit{last}(S''_1 +\!\!+ U))$. Since $|U| > 0$, we get from this that $\mathit{getGas}^-(\mathit{last}(U)) < \mathit{getGas}^-(\mathit{last}(U))$ which clearly gives a contradiction.
  2. Assume $S = U +\!\!+ S'$ for some U with $|U| > 0$. Thene by (C1) we know that (C4) $S'_1 +\!\!+ U +\!\!+ S' = S''_1 +\!\!+ S'$ and hence $S''_1 = S'_1 +\!\!+ U$. Similarly from (A3) and (B2) we know that $S'_2 +\!\!+ S = S''_2 +\!\!+ S'$ and hence $S'_2 +\!\!+ U +\!\!+ S' = S''_2 +\!\!+ S'$ and so $S''_2 = S'_2 +\!\!+ U$. Then clearly $|S''_1| > 0$ and hence by (B4) $\mathit{getGas}^-(\mathit{last}(S''_2)) < \mathit{getGas}^-(\mathit{last}(S''_1))$ and so $\mathit{getGas}^-(\mathit{last}(S'_2 +\!\!+ U)) < \mathit{getGas}^-(\mathit{last}(S'_1 +\!\!+ U))$ and since $|U| > 0$ we get from this that $\mathit{getGas}^-(\mathit{last}(U)) < \mathit{getGas}^-(\mathit{last}(U))$ which clearly gives a contradiction.

□

We can now use the notion of precedence to reason about how call stacks evolve. In particular we show the following lemma:

Lemma 5. *Let S_1, S_2, S, and U be (annotated) call stacks. If $S_1 +\!\!+ U \succ S \succ S_2 +\!\!+ U$ then there is some S_3 such that $|S_3| > 0$ and $S = S_3 +\!\!+ U$.*

Proof. Let S_1, S_2, S, and U be (annotated) call stacks and let (A) $S_1 +\!\!+ U \succ S$ and (B) $S \succ S_2 +\!\!+ U$. Towards contradiction we assume that for all call stacks S_3 with $|S_3|$ it holds that (*) $S \neq S_3 +\!\!+ U$. From (A) we know that there are call stacks V_1, S'_1, and S' such that (A1) $|S'| > 0$, (A2) $S_1 +\!\!+ U = S'_1 +\!\!+ V_1$, (A3) $S = S' +\!\!+ V_1$, and (A4) $|S'_1| > 0 \Rightarrow \mathit{getGas}^-(\mathit{last}(S')) < \mathit{getGas}^-(\mathit{last}(S'_1))$. From (B) we get that there are call stacks V_2, S'_2,

and S''' such that (B1) $|S'_2| > 0$, (B2) $S_2 +\!\!+ U = S'_2 +\!\!+ V_2$, (B3) $S = S'' +\!\!+ V_2$, and (B4) $|S''| > 0 \Rightarrow \mathit{getGas}^-(\mathit{last}(S'_2)) < \mathit{getGas}^-(\mathit{last}(S''))$.

From (A2) we know that either $U = U_1 +\!\!+ V_1$ for some U_1 with $|U_1| > 0$ or that $V_1 = U' +\!\!+ U$ for some U'. We distinguish these cases:

1. If (C1) $U = U_1 +\!\!+ V_1$ for some U_1 with $|U_1| > 0$ then by (A2) $S_1 +\!\!+ U_1 +\!\!+ V_1 = S'_1 +\!\!+ V_1$ and hence (C2) $S'_1 = S_1 +\!\!+ U_1$. Further, we know from (A3) and (B3) that (C3) $S' +\!\!+ V_1 = S'' +\!\!+ V_2$. So we know that either $V_1 = V +\!\!+ V_2$ for some V or that $V_2 = V +\!\!+ V_1$ for some V. We distinguish these cases:

   a) If $V_1 = V +\!\!+ V_2$ for some V. Then from (B2) we get $S_2 +\!\!+ U_1 +\!\!+ V +\!\!+ V_2 = S'_2 +\!\!+ V_2$ and so (C4) $S'_2 = S_2 +\!\!+ U_1 +\!\!+ V$. Further we know from (C3) that $S' +\!\!+ V +\!\!+ V_2 = S'' +\!\!+ V_2$ and hence (C5) $S'' = S' +\!\!+ V$. We make a case distinction on $|V|$

      i. If $|V| = 0$ then $V = \epsilon$. So by (C4) we have (C6) $S'_2 = S_2 +\!\!+ U_1$ and by (C5) we have (C7) $S'' = S'$ By (C2) and (A4) we know that $\mathit{getGas}^-(\mathit{last}(S')) < \mathit{getGas}^-(\mathit{last}(S_1 +\!\!+ U_1))$ and since $|U_1| > 0$ also (C8) $\mathit{getGas}^-(\mathit{last}(S')) < \mathit{getGas}^-(\mathit{last}(U_1))$. By (C6), (C7) and (B4) we have that $\mathit{getGas}^-(\mathit{last}(S_2 +\!\!+ U_1)) < \mathit{getGas}^-(\mathit{last}(S'))$ and since $|U_1| > 0$ also $\mathit{getGas}^-(\mathit{last}(U_1)) < \mathit{getGas}^-(\mathit{last}(S'))$ what contradicts (C8).

      ii. If $|V| > 0$ then by (B4) we get that $\mathit{getGas}^-(\mathit{last}(S_2 +\!\!+ U_1 +\!\!+ V)) < \mathit{getGas}^-(\mathit{last}(S' +\!\!+ V))$ and so also $\mathit{getGas}^-(\mathit{last}(V)) < \mathit{getGas}^-(\mathit{last}(V))$ which is a clear contradiction.

   b) If $V_2 = V +\!\!+ V_1$ for some V with $|V| > 0$ then from (C3) we know that $S' +\!\!+ V_1 = S'' +\!\!+ V +\!\!+ V_1$ and hence (C9) $S' = S'' +\!\!+ V$. By (B2) we know that either $V_2 = U' +\!\!+ U$ for some U' with $|U'| > 0$ or $U = U_2 +\!\!+ V_2$ for some U_2. We distinguish these cases

      - If $V_2 = U' +\!\!+ U$ for some U' with $|U'| > 0$ then by (B3) $S = S'' +\!\!+ U' +\!\!+ U$. However by (*) we can conclude that $S \neq S'' +\!\!+ U' +\!\!+ U$ what gives a clear contradiction.
      - If $U = U_2 +\!\!+ V_2$ for some U_2. Then from (B2) we get that $S_2 +\!\!+ U_2 +\!\!+ V_2 = S'_2 +\!\!+ V_2$ and hence (C10) $S'_2 = S_2 +\!\!+ U_2$. By (A2) we get that $S_1 +\!\!+ U_2 +\!\!+ V +\!\!+ V_1 = S'_1 +\!\!+ V_1$ and hence $S'_1 = S_1 +\!\!+ U_2 +\!\!+ V$. From this we get by (A4) and (C9) $\mathit{getGas}^-(\mathit{last}(S'' +\!\!+ V)) < \mathit{getGas}^-(\mathit{last}(S_1 +\!\!+ U_2 +\!\!+ V))$ and since $|V| > 0$ also $\mathit{getGas}^-(\mathit{last}(V)) < \mathit{getGas}^-(\mathit{last}(V))$ which clearly is a contradiction.

2. if (C2) $V_1 = U' +\!\!+ U$ for some U' then by (*) and, (C2) and (A3) we get that $S' +\!\!+ U' +\!\!+ U \neq S' +\!\!+ U' +\!\!+ U$ (since $|S'| > 0$ and so also $S' +\!\!+ U' > 0$. This clearly gives a contradiction.

□

Note that in particular the following corollary follows:

Lemma 6. *Let S_1, S_2, S, and U be (annotated) call stacks. Then the following statements hold*

- *If $S_1 +\!\!+ U \succcurlyeq S \succ S_2 +\!\!+ U$ then there is some S_3 such that $S = S_3 +\!\!+ U$.*
- *If $S_1 +\!\!+ U \succcurlyeq S \succ S_2 +\!\!+ U$ and $|S_1| > 0$ then there is some S_3 with $|S_3| > 0$ such that $S = S_3 +\!\!+ U$.*
- *If $S_1 +\!\!+ U \succ S \succcurlyeq S_2 +\!\!+ U$ then there is some S_3 with $|S_3| > 0$ such that $S = S_3 +\!\!+ U$.*

Proof. We prove the cases separately.

- Assume that (1) $S_1 +\!\!+ U \succcurlyeq S$ and (2) $S \succ S_2 +\!\!+ U$. We make a case distinction based on (1)
  1. Assume that (A) $S_1 +\!\!+ U \succcurlyeq S$ and (B) $S \succ S_2 +\!\!+ U$. Then we can make a case distinction on (A)
     - If $S_1 +\!\!+ U = S$ then the claim trivially holds.
     - If $S_1 +\!\!+ U \succ S$ then the statement trivially follows from Lemma 5.
  2. Assume that (A) $S_1 +\!\!+ U \succcurlyeq S$ and and (B) $S \succ S_2 +\!\!+ U$ and (C) $|S_1| > 0$ Then we can make a case distinction on (A)
     - If $S_1 +\!\!+ U = S$ then the claim trivially holds.
     - If $S_1 +\!\!+ U \succ S$ then the statement trivially follows from Lemma 5.
  3. Assume that (A) $S_1 +\!\!+ U \succ S$ and (B) $S \succcurlyeq S_2 +\!\!+ U$. Then we can make a case distinction on (B)
     - If $S = S_2 +\!\!+ U$ then we still need to show that $|S_2| > 0$. We know that $S_1 +\!\!+ U \succ S_2 +\!\!+ U$ and hence by Definition 14 we know that there are some S_1', S_2', V such that (A1) $|S_2'| > 0$ (A2) $S_2 +\!\!+ U = S_2' +\!\!+ V$, (A3) $S_1 +\!\!+ U = S_1' +\!\!+ V$, (A4) $|S_1'| > 0 \Rightarrow \mathit{getGas}^-(\mathit{last}(S_2')) < \mathit{getgas}(\mathit{last}(S_1'))$. We can conclude from (A2) and (A3), and (A4) that $V = U' +\!\!+ U$ for some U', and hence by (A2) $S_2 = S_2' +\!\!+ U'$. Since by (A1) $|S_2'| > 0$. This concludes the case.
     - If $S \succ S_2 +\!\!+ U$ then the statement trivially follows from Lemma 5.

□

We show that the precedence of call stacks decreases during execution:

Lemma 7. *Let Γ, S, S' such that $\Gamma \vDash S \rightarrow S'$. Then $S \succ S'$.*

Proof. By case distinction on the small step rules. We shortly summarize the relevant cases.

- In case of a local execution step $\Gamma \vDash s :: U \rightarrow s' :: U$ the gas decreases while the underlying call stack stays unchanged. This in particular also holds when entering an exception or halting state since exception states have the gas value -1 by the *getGas*$^-$ function.
- In case of a transaction initiating execution step $\Gamma \vDash s :: U \rightarrow s' :: s :: U$, the callee state stays unmodified. This results in S being a postfix of S' which trivially satisfies $S' \prec S$.
- In case of a transaction concluding step $\Gamma \vDash s' :: s :: U \rightarrow s'' :: U$ the callee state s is modified in that its gas is decreased. This leaves U to be the common postfix and the gas decreases from s to s'.

□

Note that, in particular, we obtain the following corollary:

Corollary. *If* $\Gamma \vDash S \rightarrow^n S'$ *then* $S \succcurlyeq S'$.

Proof. Trivially follows from Lemma 7 and the transitivity of $\prec$ (Lemma 4). □

For proving the termination small-step execution, we will use the order $\prec$. More precisely, we will perform a Noetherian induction on the $\prec$ relation. To this end, we will need to show that $\prec$ is well-founded. This property, however, does not generally hold: Intuitively, $\prec$ can be considered a lexicographical list ordering on reversed call stacks with the substantial difference that in the case that one call stack is a postfix of the other, the bigger stack is considered smaller with respect to $\prec$. This implies that there are infinite descending chains of the form $S \succ s :: S \succ s' :: s :: S \succ \ldots$. However, if we define $\prec$ on call stacks whose gas strictly decreases with each stack element, the length of such chains is limited by the amount of gas. We hence define $\succ$ on such call stacks and show that all relevant (reachable) call stacks satisfy this condition.

Definition 15 (Gas Descending Call Stacks)**.** *A call stack S is called* gas descending *if one of the following conditions hold:*

- $|S| \leq 1$
- $S = s :: s' :: S'$ *then* $\mathit{getGas}^-(s) < \mathit{getGas}^-(s')$ *and* $s' :: S'$ *is gas descending*

To prove reachable call stacks to be gas descending, we need to show a slightly stronger lemma that takes into account the gas budget allocated by the caller state. To this end, we assume a function $C_{call} \in \mathcal{S} \rightarrow \mathbb{N}$ that given a caller's state returns the gas budget given to the call (denoted c_{call} in the corresponding rules).

Definition 16 (Respect for Gas Budget)**.** *A call stack S* respects the gas budget *if one of the following conditions hold:*

- $|S| \leq 1$

- *If* $S = s :: s' :: S'$ *then* $\mathit{getGas}^-(s) \leq C_{\mathit{call}}(s')$ *and* $s' :: S'$ *respects the gas budget*

We show the following lemma:

Lemma 8 (Gas Budget Respecting Call Stacks)**.** *Let* $\Gamma \vDash s :: S \rightarrow^n S' +\!\!+ S$ *then* S' *respects the gas budget.*

Proof. By induction on n.

- If $n = 0$ then $S' = [s]$ and hence by Definition 15 S' is respects the gas budget.
- If $n > 0$ then $\Gamma \vDash s :: S \rightarrow^{n-1} S^*$ and $\Gamma \vDash S^* \rightarrow S' +\!\!+ S$. Consequently it holds that $s :: S \succcurlyeq S^* \succ S' +\!\!+ S'$. So by Lemma 6 we know that there is some S_3 such that $S^* = S^3 +\!\!+ S$ and $|S_3| > 0$. By the inductive hypothesis we know that S^3 respects the gas budget. We are hence left to show that also S' respects the gas budget. If $|S'| \leq 1$ then this trivially holds. So we consider the case where $S' = \hat{s} :: \hat{s'} :: \hat{S}$. We perform a case distinction on $\Gamma \vDash S_3 +\!\!+ S \rightarrow \hat{s} :: \hat{s'} :: \hat{S} +\!\!+ S$. We know that $S_3 = S' :: S'_3$ We distinguish the following cases:

  1. A local execution step is performed. Then $S_3 = \hat{s^*} :: \hat{s'} :: \hat{S}$ and $\mathit{getGas}^-(s^*) > \mathit{getGas}^-(\hat{s})$ (since every local execution step decreases the gas). Since S_3 respects the gas budget we know that $\mathit{getGas}^-(s^*) \leq C_{\mathit{call}}(\hat{s'})$ and that $\hat{s'} :: \hat{S}$ respects the gas budget. So we can conclude that $\mathit{getGas}^-(\hat{s}) < C_{\mathit{call}}(\hat{s'})$ and so also S' respects the gas budget.
  2. An internal transaction is initiated. Then $S_3 = \hat{s'} :: \hat{S}$ and by the definition of the small-step rules for internal transaction initiation $\mathit{getGas}^-(\hat{s}) \leq C_{\mathit{call}}(\hat{s'})$ since exactly the gas budget c_{call} is given to the callee. From S_3 respecting the gas budget we can therefore conclude that also S' respects the gas budget.
  3. An internal transaction is completed. Then $S_3 = s^* :: \hat{s''} :: \hat{s'} :: \hat{S}$ and *final* (s^*).
     - If $s^* = \mathit{EXC}$ then clearly $\mathit{getGas}^-(\hat{s''}) > \mathit{getGas}^-(\hat{s})$ since the upfront-allocated (non-zero) gas cost (c_{call} plus some additional cost) are removed from the callee state s and no money is refunded. Since S_3 respects the gas budget it is further known that $\mathit{getGas}^-(\hat{s''}) \leq C_{\mathit{call}}(\hat{s'})$ and that $\hat{s'} :: \hat{S}$ respects the gas budget. So we know that $C_{\mathit{call}}(\hat{s'}) \geq \mathit{getGas}^-(\hat{s})$ and hence also S' respects the gas budget.
     - if $s^* = \mathit{HALT}(\sigma, g, d, \eta)$ then the remaining gas g will get refunded after subtracting the upfront-allocated (non-zero) gas cost ($c = C_{\mathit{call}}(\hat{s''}) + x$ with $x > 0$). Since we know that $g \leq C_{\mathit{call}}(\hat{s''})$ we also know that $c - g > 0$ and consequently $\mathit{getGas}^-(\hat{s}) = \mathit{getGas}^-(\hat{s''}) - c + g < \mathit{getGas}^-(\hat{s''}) \leq C_{\mathit{call}}(hats')$. The last inequality follows from the inductive hypothesis, as well as that $\hat{s'}) :: \hat{S}$ respects the gas budget and hence we can conclude that S' respects the gas budget.

□

Lemma 9 (Gas Descending Call Stacks)**.** *Let* $\Gamma \vDash s :: S \rightarrow^n S' +\!\!+ S$ *then* S' *is gas descending.*

Proof. By Lemma 8 we know that S' respects the gas budget. Since we can easily check that $C_{call}(s) < \mathit{getGas}^-(s)$ for all call states s, the result follows from a simple induction on the size of S'. □

Lemma 10 (Well-foundedness of Precedence)**.** *$\prec$ is a well-founded relation on gas descending call stacks ($\{S \in \mathbb{S} \mid |S| > 0 \ \wedge \ S$ is gas descending$\}$.*

Proof. We assume towards contradiction that there is an infinite descending chain $S_1 \succ S_2 \succ \cdots$. We show that from this we can construct an infinite descending chain of $<$ on $\mathbb{N}$ what clearly yields a contradiction since $<$ on $\mathbb{N}$ is well-founded. We construct the following measure on gas descending (non-empty) call stacks:

$$m' \in \mathcal{L}(\mathbb{N}) \to \mathbb{N}$$
$$m'(\mathit{ns}) = \begin{cases} 0 & \mathit{ns} = [0] \\ n! & \mathit{ns} = [n] \ \wedge \ n > 0 \\ (n-1)! & n + m'(\mathit{ns}') = n :: \mathit{ns}' \ \wedge \ |\mathit{ns}| > 0 \end{cases}$$

$$m \in \{S \in \mathbb{S} \mid |S| > 0 \ \wedge \ S \text{ is gas descending}\} \to \mathbb{N}$$
$$m(S) = \mathit{rev}(\mathit{map}(\lambda s.\, (\mathit{getGas}^-(s) + 2) * 2, S))$$

Note that here $\mathit{rev}(\cdot)$ and $\mathit{map}(\cdot,\cdot)$ denote the standard notions of reversing a list and applying a function to each element of a list. We state some general properties about m' that can be all proven by simple induction:

- $|ls| > 0 \Rightarrow m'(l_1 +\!\!+ l_2) = m'(l_1 +\!\!+ [0]) + m'(l_2)$
- $m'(n :: l) \leq n!$

- We first show that for each $S \in \{S \in \mathbb{S} \mid S \text{ is gas descending}\}$ that $m(S) \geq 0$. By induction on the size of S.
  1. Let $S = [s]$. Then $m([s]) = ((\mathit{getGas}^-(s) + 2) * 2)! \geq 0$, since $\mathit{getGas}^-(s) \geq -1$.
  2. Let $S = S' \cdot [s]$ and $|S'| > 0$. Let in the following be $f = \lambda s.\, (\mathit{getGas}^-(s) + 2) * 2$ Then $m(S) = m'(f(s) :: \mathit{rev}(\mathit{map}(f, S'))) = (f(s) - 1)! + m'(\mathit{rev}(\mathit{map}(f, S')) = (f(s) - 1)! + m(S') \geq 0$ since by the inductive hypothesis $m(S') \geq 0$ and also $(f(s) - 1)! \geq 0$ (since $\mathit{getGas}^-(s) \geq -1$).

- We now show that if $S \prec S'$ then also $m(S) < m(S')$. Let $S \prec S'$. By definition 14, hence there are some S_1, S', U such that (A1) $|S_1| > 0$, (A2) $S = S_1 +\!\!+ U$, (A3) $S' = S_2 +\!\!+ U$, (A4) $|S_2| > 0 \Rightarrow \mathit{getGas}^-(\mathit{last}(S_1)) < \mathit{getGas}^-(\mathit{last}(S_2))$. We first perform a case distinction on $|S_2|$. Let in the following be $f = \lambda s.\, (\mathit{getGas}^-(s) + 2) * 2$

1. If $|S_2| = 0$ then we know that $|U| > 0$ (since $|S'| > 0$) and so $U = u :: U'$ for some u, U'. Further we know that

$$\begin{aligned} m(S) &= m'(rev(map(f, S_1 +\!\!+ U))) \\ &= m'(rev(map(f, U) +\!\!+ rev(map(f, S_1))) \\ &= m'(rev(map(f, U)) +\!\!+ [0]) + m'(rev(map(f, S_1))) \\ &= m'(rev(map(f, u :: U')) +\!\!+ [0]) + m'(rev(map(f, S_1))) \\ &= m'(rev(map(f, U')) +\!\!+ [f(u), 0]) + m'(rev(map(f, S_1))) \\ &= m'(rev(map(f, U')) +\!\!+ [0]) + m'([f(u), 0]) + m'(rev(map(f, S_1))) \\ &= m'(rev(map(f, U')) +\!\!+ [0]) + (f(u) - 1)! + m'(rev(map(f, S_1))) \end{aligned}$$

   Further we have

$$\begin{aligned} m(S') &= m'(rev(map(f, U))) \\ &= m'(rev(map(f, u :: U'))) \\ &= m'(rev(map(f, U)) +\!\!+ [f(u)]) \\ &= m'(rev(map(f, U)) +\!\!+ [0]) + m'([f(u)]) \\ &= m'(rev(map(f, U)) +\!\!+ [0]) + (f(u))! \end{aligned}$$

   It is hence sufficient to show that $(f(u) - 1)! + m'(rev(map(f, S_1))) < (f(u))!$ Since all elements in $map(f, S_1)$ need to be smaller or equal than $f(u) - 2$, we also know that $m'(rev(map(f, S_1))) \leq (f(u) - 2)!$. Since we know that $(n - 1)! + (n - 2)! < n!$ for all $n > 0$ we can conclude that

$$\begin{aligned} (f(u) - 1)! + m'(rev(map(f, S_1))) &\leq (f(u) - 1)! + (f(u) - 2)! \\ &< (f(u))! \end{aligned}$$

2. If $|S_2| > 0$ then we know that $S_1 = S_1' +\!\!+ s_1$ for some S_1' and s_1' and $S_2 = S_2' +\!\!+ s_2$ for some S_2', s_2 and $getGas^-(s_1) < getGas^-(s_2)$. Hence we know that

$$\begin{aligned} m(S) &= m'(rev(map(f, S_1 +\!\!+ U))) \\ &= m'(rev(map(f, U) +\!\!+ rev(map(f, S_1))) \\ &= m'(rev(map(f, U)) +\!\!+ [0]) + m'(rev(map(f, S_1))) \\ &= m'(rev(map(f, U)) +\!\!+ [0]) + m'(f(s_1) :: rev(map(f, S_1'))) \end{aligned}$$

   and

$$\begin{aligned} m(S') &= m'(rev(map(f, S_2 +\!\!+ U))) \\ &= m'(rev(map(f, U) +\!\!+ rev(map(f, S_2))) \\ &= m'(rev(map(f, U)) +\!\!+ [0]) + m'(rev(map(f, S_2))) \\ &= m'(rev(map(f, U)) +\!\!+ [0]) + m'(f(s_2) :: rev(map(f, S_2'))) \end{aligned}$$

   It is hence sufficient to show that $m'(f(s_1) :: rev(map(f, S_1'))) < m'(f(s_2) :: rev(map(f, S_2')))$ We do another case distinction on the lengths of S_1' and S_2':

a) If $|S_1'| = |S_2'|$ then

$$\begin{aligned} m'(f(s_1) :: rev(map(f, S_1'))) &= m'([f(s_1)]) \\ &= (f(s_1))! \\ &< (f(s_2))! \\ &= m'([f(s_2)]) \\ &= m'(f(s_2) :: rev(map(f, S_2'))) \end{aligned}$$

Note that $(f(s_1))! < (f(s_2))!$ since $f(s_1) < f(s_2)$ (since $getGas^-(s_1) < getGas^-(s_2)$) and $f(s_1) \geq 2$.

b) If $|S_1'| = 0$ and $|S_2'| > 0$ then

$$\begin{aligned} m'(f(s_1) :: rev(map(f, S_1'))) &= m'([f(s_1)]) \\ &= (f(s_1))! \\ &< (f(s_2) - 1)! \\ &\leq (f(s_2) - 1)! + m'(rev(map(f, S_2'))) \\ &= m'([f(s_2), 0]) + m'(rev(map(f, S_2'))) \\ &= m'(f(s_2) :: rev(map(f, S_2'))) \end{aligned}$$

Note that $(f(s_1))! < (f(s_2) - 1)!$ since $f(s_1) < f(s_2) - 1$ (and $f(s_1) \geq 1$). More precisely:

$$\begin{aligned} & f(s_1) < f(s_2) - 1 \\ \Leftrightarrow\ & (getGas^-(s_1) + 2) * 2 < (getGas^-(s_2) + 2) * 2 - 1 \\ \Leftrightarrow\ & 2 * getGas^-(s_1) + 4 < 2 * getGas^-(s_2) + 3 \\ \Leftrightarrow\ & 2 * getGas^-(s_1) + 1 < 2 * getGas^-(s_2) \\ \Leftrightarrow\ & 2 * getGas^-(s_1) + 2 \leq 2 * getGas^-(s_2) \\ \Leftrightarrow\ & 2 * (getGas^-(s_1) + 1) \leq 2 * getGas^-(s_2) \\ \Leftrightarrow\ & getGas^-(s_1) + 1 \leq getGas^-(s_2) \\ \Leftrightarrow\ & getGas^-(s_1) < getGas^-(s_2) \end{aligned}$$

3. If $|S_1'| > 0$ and $|S_2'| = 0$ then

$$\begin{aligned} m'(f(s_1) :: rev(map(f, S_1'))) &= m'([f(s_1), 0]) + m'(rev(map(f, S_1'))) \\ &= (f(s_1) - 1)! + m'(rev(map(f, S_1'))) \\ &\leq (f(s_1) - 1)! + (f(s_1))! \\ &< (f(s_1) + 1)! \\ &\leq (f(s_2))! \\ &= m'([f(s_2)]) \\ &= m'(f(s_2) :: rev(map(f, S_2'))) \end{aligned}$$

Note that $(f(s_1)+1)! \leq (f(s_2))!$ since $f(s_1)+1 \leq f(s_2)$. In particular:

$$\begin{aligned} & f(s_1)+1 \leq f(s_2) \\ \Leftrightarrow\ & (\mathit{getGas}^-(s_1)+2)*2+1 \leq (\mathit{getGas}^-(s_2)+2)*2 \\ \Leftrightarrow\ & 2*\mathit{getGas}^-(s_1)+5 \leq 2*\mathit{getGas}^-(s_2)+4 \\ \Leftrightarrow\ & 2*\mathit{getGas}^-(s_1)+1 \leq 2*\mathit{getGas}^-(s_2) \\ \Leftrightarrow\ & 2*\mathit{getGas}^-(s_1) < 2*\mathit{getGas}^-(s_2) \\ \Leftrightarrow\ & \mathit{getGas}^-(s_1) < \mathit{getGas}^-(s_2) \end{aligned}$$

4. If $|S_1'| > 0$ and $|S_2'| > 0$ then

$$\begin{aligned} m'(f(s_1) :: \mathit{rev}(\mathit{map}(f, S_1'))) &= m'([f(s_1), 0]) + m'(\mathit{rev}(\mathit{map}(f, S_1'))) \\ &= (f(s_1)-1)! + m'(\mathit{rev}(\mathit{map}(f, S_1'))) \\ &\leq (f(s_1)-1)! + (f(s_1))! \\ &< (f(s_2)-1)! \\ &\leq (f(s_2)-1)! + m'(\mathit{rev}(\mathit{map}(f, S_2'))) \\ &= m'([f(s_2), 0]) + m'(\mathit{rev}(\mathit{map}(f, S_2'))) \\ &= m'(f(s_2) :: \mathit{rev}(\mathit{map}(f, S_2'))) \end{aligned}$$

Note that $(f(s_1)-1)! + (f(s_1))! < (f(s_2)-1)!$ since $(f(s_1)-1)! + (f(s_1))! < (f(s_1)+1)!$ (and $f(s_1)+1 \leq 3$) and $(f(s_1)+1)! \leq (f(s_2)-1)!$ which holds since $(f(s_1)+1)! < (f(s_2)-1)!$. More precisely

$$\begin{aligned} & (f(s_1)+1)! < (f(s_2)-1)! \\ \Leftrightarrow\ & (\mathit{getGas}^-(s_1)+2)*2+1 \leq (\mathit{getGas}^-(s_2)+2)*2-1 \\ \Leftrightarrow\ & 2*\mathit{getGas}^-(s_1)+5 \leq 2*\mathit{getGas}^-(s_2)3 \\ \Leftrightarrow\ & 2*\mathit{getGas}^-(s_1)+2 \leq 2*\mathit{getGas}^-(s_2) \\ \Leftrightarrow\ & 2*(\mathit{getGas}^-(s_1)+1) \leq 2*\mathit{getGas}^-(s_2) \\ \Leftrightarrow\ & (\mathit{getGas}^-(s_1)+1) \leq \mathit{getGas}^-(s_2) \\ \Leftrightarrow\ & \mathit{getGas}^-(s_1) < \mathit{getGas}^-(s_2) \end{aligned}$$

□

Lemma 11 (Termination)**.** *Let Γ be a transaction environment and S be a call stack. Then there exists s and π such that final (s) and $\Gamma \vDash S \xrightarrow{\pi}{}^{*} [s]$.*

Proof. By Noetherian induction on $\succ$. By Lemma 2 we know that either

1. $S = [s]$ and *final* (s). In this case the claim trivially holds.

2. $\Gamma \vDash S \xrightarrow{a} S'$. In this case we know by Lemma 7 that $S \succ S'$. Consequently we can use the inductive hypothesis for S' and hence know that $\Gamma \vDash S' \xrightarrow{\pi}{}^{*} [s]$ and so also $\Gamma \vDash S \xrightarrow{a \cdot \pi}{}^{*} [s]$.

□

Lemma 12 (Local Termination)**.** *Let Γ be a transaction environment, S be a call stack, and s an execution state. Then there exists s' and π such that final (s') and $\Gamma \vDash s +\!\!+ S \xrightarrow{\pi}{}^{*} s' +\!\!+ S$.*

Proof. By Noetherian induction on $\succ$. By Lemma 3 we know that either

1. *final* (s). In this case the claim trivially holds.
2. $\Gamma \vDash s :: S \xrightarrow{a} S' +\!\!+ S$ for some S' with $1 \leq |S'| \leq 2$. In this case we know by Lemma 7 that $s :: S \succ S' +\!\!+ S$. Further we know that either
   a) $S' = [s'']$. In this case we can use the inductive hypothesis to conclude that $\Gamma \vDash s :: S \xrightarrow{a \cdot \pi}{}^{*} s' :: S$ for some s' such that *final* (s').
   b) $S' = [s^1, s]$. In this case we can use the inductive hypothesis to conclude that $\Gamma \vDash s^1 :: s :: S \xrightarrow{\pi_1}{}^{*} s^2 :: s :: S$ for some s^2 such that *final* (s^2). By the definition of the small-step semantics we further know that $\Gamma \vDash s^2 :: s :: S \rightarrow s^3 :: S$ and consequently also by Lemma 7 that $s :: S \succ s^3 :: S$ so that we can use the inductive hypothesis again to conclude that $\Gamma \vDash s^3 +\!\!+ S \xrightarrow{\pi_2}{}^{*} s' +\!\!+ S$ for some s' such that *final* (s'). This concludes the proof since in sum we have that $\Gamma \vDash s +\!\!+ S \xrightarrow{a \cdot \pi_1 \cdot \pi_2}{}^{*} s' +\!\!+ S$.

□

Using Lemma 5 we can formally characterize how call stacks are preserved during execution:

Lemma 13 (Call Stack Preservation during Execution)**.** *Let (Γ, S) be a configuration such that $\Gamma \vDash U +\!\!+ S \rightarrow^{n} U' +\!\!+ S$. Then the following properties hold:*

- *if $U' = \epsilon$ then $U = \epsilon$*
- *if $U = \epsilon$ and $U' \neq \epsilon$ then there are $s \in \mathcal{S}$, $c \in \mathcal{C}$ such that $\Gamma \vDash S \rightarrow s_c :: S$ and $\Gamma \vDash s_c :: S \rightarrow^{n-1} U' +\!\!+ S$.*
- *if $U \neq \epsilon$ and $U' \neq \epsilon$ and $\Gamma \vDash U +\!\!+ S \rightarrow^{m} S'$ and $\Gamma \vDash S' \rightarrow^{n-m} U' +\!\!+ S$ for $0 \leq m \leq n$ then there exists U'' such that $|U''| > 0$ and $S' = U'' +\!\!+ S$*

Proof. We show the three properties in separation:

- We distinguish the cases $n = 0$ and $n > 0$.

  - If $n = 0$ then $U = U'$ and hence the property trivially holds.
  - Let $n > 0$ and $U' = \epsilon$. By Lemma 7 we know that $U +\!\!+ S \succ S$. Hence by Definition 14 we have that there are some V, S_1, S_2 such that (A1) $|S_1| > 0$, (A2) $U +\!\!+ S = S_2 +\!\!+ V$, (A3) $S = S_1 +\!\!+ V$, and (A4) $|S_2| > 0 \Rightarrow \mathit{getGas}^-(\mathit{last}(S_1)) < \mathit{getGas}^-(\mathit{last}(S_2))$. By (A2) and (A3) we get that $U +\!\!+ S_1 +\!\!+ V = S_2 +\!\!+ V$ and hence $S_2 = U +\!\!+ S_1$. So from (A4) and (A1) we then know that $|S_2| > 0$ and so also $\mathit{getGas}^-(\mathit{last}(S_1)) < \mathit{getGas}^-(\mathit{last}(U +\!\!+ S_1))$ and as a consequence given (A1) also $\mathit{getGas}^-(\mathit{last}(S_1)) < \mathit{getGas}^-(\mathit{last}(S_1))$ which gives a clear contradiction.

- To satisfy the premise, it needs to holds that $n > 0$. Let $U = \epsilon$ and $U' \neq \epsilon$. Then by the assumption we know that $\Gamma \vDash S \rightarrow S'$ and $\Gamma \vDash S' \rightarrow^{n-1} U' +\!\!+ S$. So by Lemma 7, we know that $S \succ S' \succcurlyeq U' +\!\!+ S$ and by Lemma 6 we have that there is some S_3 with $|S_3| > 0$ such that $S' = S' +\!\!+ S$. Consequently we have that $\Gamma \vDash S \rightarrow S' +\!\!+ S$. By case distinction on the small step rules we know that this means that $S_3 = [s_c]$ for some annotated execution state s_c. This closes the case.

- Let $U \neq \epsilon$, $U' \neq \epsilon$, and $\Gamma \vDash U +\!\!+ S \rightarrow^m S'$ and $\Gamma \vDash S' \rightarrow^{n-m} U' +\!\!+ S$ for $0 \leq m \leq n$. Then we know by Lemma 7 that $U +\!\!+ S \succcurlyeq S'$ and $S' \succcurlyeq U' +\!\!+ S$. We distinguish two cases:
  - Let (A) $U +\!\!+ S = S'$. Then the property trivially holds since by assumption $|U| > 0$.
  - If $U +\!\!+ S \succ S'$ the property immediately follows from Lemma 6.

□

Another observation is that the influence of the call stack on the contract's execution is limited to the size of the call stack. Depending on the call stack size, an error in the top-level execution might occur due to exceeding the call stack limit.

Formally, we capture this property in the following lemma:

Lemma 14 (Call Stack Indifference up to Size)**.** *Let s be an execution state, Γ a transaction environment and let S, S' and U be call stacks such that $|S| = |U|$. Then it holds for all $n \in \mathbb{N}$ that*

$$\Gamma \vDash s :: S \rightarrow^n S' +\!\!+ S \Leftrightarrow \Gamma \vDash s :: U \rightarrow^n S' +\!\!+ U$$

Proof. By induction on n. Since both directions are fully symmetric we only show one direction.

1. Let $n = 0$. Then $S' = [s]$ and hence the claim trivially holds.
2. Let $n > 0$. Then (A1) $\Gamma \vDash s :: S \rightarrow^{n-1} S^*$ and (A2) $\Gamma \vDash S^* \rightarrow S' +\!\!+ S$. Then by Lemma 6 we know that there is some S_3 such that $|S_3| > 0$ and $S^* = S_3 +\!\!+ S$. By the inductive hypothesis and (A1) we hence know that also $\Gamma \vDash s :: S \rightarrow^{n-1} S_3 +\!\!+ U$. Since $|S_3| > 0$ we know that $S_3 = s^3 :: S_3'$ for some s^3 and S_3'. We perform a case distinction on the execution step performed in (A2)

a) If a local execution step is performed then $S' = s^4 :: S'_3$ and correspondingly also $\Gamma \vDash s^3 :: S'_3 +\!\!+ U \rightarrow s^4 :: S'_3 +\!\!+ U$

b) If an internal transaction is initiated then $S' = s^4 :: S_3$ and consequently also $\Gamma \vDash s^3 :: S'_3 +\!\!+ U \rightarrow s^4 :: s^3 :: S'_3 +\!\!+ U$.

c) If an internal transaction is completed then we know that $S'_3 = s^4 :: S_4$ and $S' = s^5 :: S_4$. So consequently also $\Gamma \vDash s^3 :: s^4 :: S_4 +\!\!+ U \rightarrow s^5 :: S_4 +\!\!+ U$.

□

A.6.1 Forms of States

We introduce the notion of a *call state* for characterizing those states that invoke internal transactions.

Definition 17 (Call States)**.** *A regular execution state s is a call state if for all Γ and S, such that $|S| < 1024$, it holds that $\Gamma \vDash s :: S \rightarrow s' :: s :: S$ for some s'.*

Intuitively, an execution state is a call state if it satisfies all preconditions for a transaction initiating instruction.

Lemma 15. *Let s be a call state. Then $s.\iota.\mathsf{code}[s.\mu.\mathsf{pc}] \in Inst_{call}$.*

Proof. Proof by simple case distinction on the small step rules. □

Lemma 16 (Alternative Characterization of Call States)**.** *A regular execution state s is a call state there exist Γ and S, such that $|S| < 1024$ and it holds that $\Gamma \vDash s :: S \rightarrow s' :: s :: S$ for some s'.*

Proof. To show the equivalence of the two characterizations, we show the two directions in separation.

1. Let s be a call state Definition 17. Pick an arbitrary transaction environment and $S = \epsilon$. Then clearly $\Gamma \vDash s :: S \rightarrow s' :: s :: S$ for some s' by definition.

2. Let s be an execution state and $\Gamma \vDash s :: S \rightarrow s' :: s :: S$ for some s' for some Γ, S and s' such that $|S| < 1024$. Let now Γ' and S' be an arbitrary transaction environment and call stack such that $|S'| < 1024$. We show that also $\Gamma' \vDash s :: S' \rightarrow s' :: s :: S'$. By Lemma 15 we know that $s.\iota.\mathsf{code}[s.\mu.\mathsf{pc}] \in Inst_{call}$. By examination of the small-step rules we can immediately see that all rules that apply in this case are independent from the transaction environment and the call stack up to the fact that the call stack limit is reached. This case however is excluded since $|S| < 1024$ and $|S'| < 1024$. Hence we can conclude that the very same internal transaction initiating small step can be performed for S' and Γ' as it can for S and Γ.

□

In a regular execution all elements of a call stack but its top element are call states.

Lemma 17. *Let* $\Gamma \vDash s :: S \rightarrow^n s' :: S' +\!\!+ S$*, then every execution state* $s'' \in S'$ *is a call state.*

Proof. By induction on n.

1. If $n = 0$ then $s :: S = s' :: S' +\!\!+ S$. In this case $S' = \epsilon$ and hence the claim follows trivially.

2. If $n > 0$ then (A1) $\Gamma \vDash s :: S \rightarrow S^*$ and (A2) $\Gamma \vDash S^* \rightarrow^{n-1} s' :: S' +\!\!+ S$. In this case we know due to Lemma 6 that there exists some S_3 such that $|S_3| > 0$ and $S^* = S_3 +\!\!+ S$. We perform a case distinction on the small step in (A1)

   - In case of a local execution step we have that $S^* = s'' :: S$. Consequently the claim immediately follows from the inductive hypothesis.
   - In case of initiating an internal transaction we have have that $S^* = s'' :: s :: S$ and so by definition s is a call state. By the inductive hypothesis we know that all $s^* \in S$ are call states what shows the claim.

   Note that we can exclude the case of an internal transaction completing a small step since we know that $S^* = S_3 +\!\!+ S$ and hence the underlying call stack can't be modified as it would happen in this case.

□

Whenever some configuration is reachable, the execution before stepped through the call states on the call stacks. This property is formally captured by the following lemma:

Lemma 18. *Let* $\Gamma \vDash S \rightarrow^n (S_1 +\!\!+ S_2) +\!\!+ S$*. Then there exists some* $m \in \mathbb{N}$ *such that* $\Gamma \vDash S \rightarrow^m S_2 +\!\!+ S$ *and* $\Gamma \vDash S_2 +\!\!+ S \rightarrow^{n-m} (S_1 +\!\!+ S_2) +\!\!+ S$*.*

Proof. Note that if $|S_1| = 0$ the claim trivially holds for $m = n$. We hence in the following assume $|S_1| > 0$, so $S_1 = exstate^1 :: S_1'$ for some s^1 and S_1'. By induction on n:

1. If $n = 0$ then $S = (S_1 +\!\!+ S_2) +\!\!+ S$ and hence $S_1 = S_2 = \epsilon$. So the claim trivially follows for $m = 0$.

2. if $n > 0$ then (A1) $\Gamma \vDash S \rightarrow^{n-1} S^*$ and (A2) $\Gamma \vDash S^* \rightarrow (S_1 +\!\!+ S_2) +\!\!+ S$. By Lemma 6 we know that there is some S_3 such that $S^* = S^3 +\!\!+ S$ and $|S_3| > 0$. So in particular we know that $S^* = s^3 :: S_3'$ for some s' and S_3'. We make a case distinction on the small step performed in (A2)

a) In case of a local execution step we know that $s^1 :: S_1' \cdot S_2 = s^4 :: S_3'$. And so also $S_3 = (s^3 :: S_1') +\!\!+ S_2$. So we can apply the inductive hypothesis to conclude the case.

b) In case of the initiation of an internal transaction we know that $s^1 :: S_1' \cdot S_2 = s^4 :: s^3 :: S_3'$ and so also $S_3 = S_1' +\!\!+ S_2$. So again the claim follows by the inductive hypothesis

c) In case of completing an internal transaction we know that $S_3' = s^4 :: S_3''$ and $s^1 :: S_1' \cdot S_2 = s^5 :: S_3''$. So we know that $S_3 = ([s^3, s^4] +\!\!+ S_1') +\!\!+ S_2$ and hence again the claim follows by the inductive hypothesis.

□

We introduce the notion of an initial execution state. Whenever a contract is called or created, the execution of the contract or initialization codes starts in a fresh machine state at program counter zero with the local memory initialized to all zeros, the machine stack being empty, and the number of active words set to zero. We characterize execution states with machine states of this form as initial:

Definition 18 (Initial Execution States)**.** *An execution state $s \in \mathcal{S}$ is called initial if it is of the form $((g, 0, \lambda x.\, 0, 0, \epsilon), \iota, \sigma)$ for some $g \in \mathbb{N}$, $\iota \in I$ and $\sigma \in \Sigma$.*

We can state the property that the execution of EVM bytecode always starts at an initial state. Or more precisely: An execution of a contract c leading to a reachable execution state must have passed an initial state of c before.

Lemma 19. *Let $\Gamma \vDash s :: S \rightarrow^n s' :: S' +\!\!+ S$ such that $n > 0$ and $|S'| > 0$. Then either*

- *there is an initial execution s^i state such that $\Gamma \vDash s :: S \rightarrow^m s^i :: S' +\!\!+ S$ and $\Gamma \vDash s^i :: S' +\!\!+ S \rightarrow^{n-m} s' :: S' +\!\!+ S$ for some $0 < m \leq n$, or*
- *$s' = EXC$ and $\Gamma \vDash s :: S \rightarrow^{n-1} S' +\!\!+ S$ and $\Gamma \vDash S' +\!\!+ S \rightarrow s' :: S' +\!\!+ S$*

Proof. By induction on n

1. If $n = 1$ then we know by a case distinction on the small step rules that $S' = [s]$ and that either $s' = EXC$ (since the transaction initiation failed at call time) or s' is an initial state. This concludes the case.

2. If $n > 1$ then (A1) $\Gamma \vDash s :: S \rightarrow^{n-1} S^*$ and (A2) $\Gamma \vDash S^* \rightarrow s' :: S' +\!\!+ S$ by 6 we know that $S^* = S_3 +\!\!+ S$ for some S_3 with $|S_3| > 0$. Hence in particular we have that $S_3 = s^3 :: S_3'$ for some s^3 and S_3'. We make a case distinction on the size of S_3':

   a) If $|S_3'| = 0$ then we know from (A2) that $S' = [s^3]$ and the claim follows by the the same reasoning as done in the base case.

b) If $|S'_3| > 0$ then we do a case distinction on the execution step performed in (A2)

- If a local execution step is performed then we have that $S' = S'_3$ and s^3. Hence by the inductive hypothesis we know that $\Gamma \vDash s :: S \rightarrow^m s^i :: S' +\!\!+ S$ and $\Gamma \vDash s^i :: S' +\!\!+ S \rightarrow^{n-1-m} s^3 :: S' +\!\!+ S$ for some $m \leq n-1$. and so consequently the first case of the claim holds .
- If an internal transaction is initiated then we know that $S' = s^3 :: S'_3$ and $s^3 \neq \mathit{EXC}$. The claim hence follows by the same reasoning as performed in the base case.
- If an internal transaction is completed then we know that $S'_3 = s^4 :: S_4$ and $S' = S_4$ and so also $|S_4| > 0$. By Lemma 18 we know that there is some $m' \leq n-1$ such that $\Gamma \vDash s :: S \rightarrow^{m'} s^4 :: S_4 +\!\!+ S'$ We can hence apply the inductive hypothesis for m' and obtain two possible cases:
  i. $\Gamma \vDash s :: S \rightarrow^{m''} s^i :: S_4 +\!\!+ S$ and $\Gamma \vDash s^i :: S_4 \cdot S \rightarrow^{m'-m''} s^4 :: S_4 +\!\!+ S$ for some $0 < m'' \leq m'$. This concludes the case.
  ii. $s^4 = \mathit{EXC}$. This case leads to a contradiction since by Lemma 17 we know that s^4 needs to be a call state and hence cannot be *EXC*.

□

This lemma states that each reachable execution state started in an initial state (unless it is one of the special cases of a transaction-initiating transaction that failed at call time and for this reason pushed *EXC* immediately to the stack.

Lemma 20. *Let* $(\Gamma, s :: S)$ *be a reachable configuration and* s *be a regular execution state. Then there exists an initial execution state* s^i *such that* $(\Gamma, s^i :: S)$ *is a reachable configuration and*

$$\Gamma \vDash s^i :: S \rightarrow^* s :: S$$

Proof. Since $(\Gamma, s_c :: S)$ is reachable, we know that there exists some s^i, c^i such that $(\Gamma, {s^i}_{c^i}) = \mathit{initializeT}(T, \sigma, H, g)$ for some transaction T, global state σ, block header H and gas g and $\Gamma \vDash {s^i}_{c^i} :: \epsilon \rightarrow^* s_c :: S$. We do a case distinction on the call stack S.

- If $S = \epsilon$ then we are done, since by definition of *initializeT*, s^i is initial
- If $S = {s'}_{c'} :: S'$, then the claim immediately follows from Lemma 19.

□

A.6.2 Hash Collisions During Contract Creation

In the original EVM semantics it was possible that an already deployed contract was changed since due to a hash collision, a new contract would be created at the very same address. Accounting for this behavior would undermine all trust assumptions on existing contracts. However, hash collisions only occur with a negligible probability what makes it a reasonable assumption to exclude such collisions for a (small) fixed set of contracts.

Definition 19 (Contracts for Addresses)**.** *The contracts for a set of addresses $\mathcal{A}_T$ in an execution state s (written $\mathit{codes}_{\mathcal{A}_T}(s)$) are defined as follows:*

$$\begin{aligned}\mathit{codes}_{\mathcal{A}_T}(s) =& \{(a, s.\sigma(a).\mathit{code}) \mid a \in \mathcal{A}_T \ \wedge \ s.\sigma(a) \neq \bot \ \wedge \ s \neq \mathit{EXC}\} \\ & \cup \{(a, \bot) \mid a \in \mathcal{A}_T \ \wedge \ s.\sigma(a) \neq \bot \ \wedge \ s \neq \mathit{EXC}\}\end{aligned}$$

Definition 20 (Collision-free Execution)**.** *A (n-step) execution $\Gamma \vDash s :: S \ \rightarrow^n \ S' +\!\!+ S$ is collision-free for the set $\mathcal{A}_T$ of addresses if for all $m \leq n$ and all call stacks S'' such that $\Gamma \vDash s :: S \ \rightarrow^m \ S'' +\!\!+ S$ it holds that for all $s' \in S'$ that*

$$\mathit{codes}_{\mathcal{A}_T}(s') = \mathit{codes}_{\mathcal{A}_T}(s)$$

Intuitively, in a collision-free execution for $\mathcal{A}_T$ the code of trusted addresses does not change in the course of the execution.

A.6.3 Annotations

As previously discussed, we assume execution states to be annotated with the contracts that they are currently executing. We formally define contract annotations in the following.

Definition 21 (Contract Annotations)**.** *Let $\Gamma \vDash s :: S \ \rightarrow^n \ s' :: S' +\!\!+ S$ be a small-step execution starting from regular execution state s. Then execution state s' is annotated with contract c' (written $s'_{c'}$) if one of the following holds*

1. *isRegular(s') and $a = s'.\iota.\mathit{actor}$ and $c' = (a, s.\sigma(a).\mathit{code})$*
2. *final (s') and $\Gamma \vDash s :: S \ \rightarrow^{n-1} \ s'' :: S' +\!\!+ S$ and $\Gamma \vDash s'' :: S' +\!\!+ S \ \rightarrow \ s' :: S' +\!\!+ S$ for some regular execution state s'' such that $s''_{c'}$*
3. *$s' = \mathit{EXC}$ and $s' = s'' :: S''$ for some s'' and S'' such that $s''_{c'}$, and $\Gamma \vDash s :: S \ \rightarrow^{n-1} \ s'' :: S'' +\!\!+ S$ and $\Gamma \vDash s'' :: S'' +\!\!+ S \ \rightarrow \ s' :: s'' :: S'' +\!\!+ S$*

Note that by this definition, the annotation of every execution state is well-defined since the only way of entering a final state is either to step there from a regular execution state or to fail at call time when performing a transaction initiating instruction. These cases are covered by conditions (2) and (3).

These annotations need to be consistent with the current execution state in the sense that they correspond to the active account of the execution state and that they present a valid contract in the global state.

Definition 22 (Annotation Consistency)**.** *An execution state s is consistent with contract annotation c if the following two conditions hold*

1. *isRegular$(s) \implies s.\iota.\mathit{actor} = c.\mathit{addr}$*

2. $\mathit{isRegular}(s) \vee \mathit{isHalt}(s) \implies s.\sigma(c.\mathit{addr}).\mathit{code} = c.\mathit{code}$

The consistency of annotations is preserved over execution.

Lemma 21 (Preservation of Annotation Consistency)**.** *Let s be consistent with c and $\Gamma \vDash s_c :: S \rightarrow^n S' +\!\!+ S$ for some Γ, S, and S'. Then for all ${s'}_{c'} \in S'$ it holds that s' is consistent with c'.*

Proof. By induction on n.

1. If $n = 0$ then $s_c :: S = S' +\!\!+ S$ and hence $S' = [s_c]$. Consequently the claim trivially holds.
2. If $n > 0$ then (A1) $\Gamma \vDash s_c :: S \rightarrow^{n-1} S^*$, and (A2) $\Gamma \vDash S^* \rightarrow S' +\!\!+ S$. By 6 we know that there is some S_3 such that $|S_3| > 0$ and $S^* = S_3 +\!\!+ S$. By the inductive hypothesis we hence know that for all ${s'}_{c'} \in S_3$ we have that s' is consistent with c'. Since $|S_3| > 0$ we know that $S_3 = {s^3}_{c''} :: S'_3$ for some s^3 and S'_3. We make a case distinction on the small step performed in (A2):
   a) In case of a local execution step we know that $S' = {s^4}_{c''} :: S'_3$ for some s_4 that has the same execution environment and account codes as s_3. Hence from s^3 being consistent with c'' we can conclude that s^4 is consistent with c'' what concludes the case.
   b) In case of the initiation of an internal transaction we know that $S' = {s^4}_{c'''} :: {s^3}_{c''} :: S'_3$. By the definitions of the corresponding small step rules we know that in this case the annotation is computed by looking up the code of the address being set as $s^4.\iota.\mathsf{actor}$, and hence trivially s^4 is consistent with c'''. This concludes the case.
   c) In case of completing an internal transaction we have that $S'_3 = {s^4}_{c'''} :: S''_3$ and $S' = {s^5}_{c''''} :: S''_3$. We know that s^5 is consistent with c'''' since, in case of an exception, the global state and the execution environment stay unchanged, and otherwise, the annotation is updated according to the global state in the halting state.

□

Contract annotations reflect the active contract that is executed. The active account of an execution state cannot be changed during execution. Formally, this is stated by the following lemma:

Lemma 22 (Execution Environment Persistence)**.** *Let $\Gamma \vDash S_1 +\!\!+ s :: S \rightarrow^n S_2 +\!\!+ s' :: S$ such that s, s' are regular execution states. Then it holds that $s.\iota = s'.\iota$.*

Proof. Proof by induction on the number n of small steps.

1. For $n = 0$ the property trivially holds since $s = s'$.

2. For $n > 0$ we know that (A1) $\Gamma \vDash S_1 +\!\!+ s :: S \ \rightarrow^{n-1} \ S^*$ and (A2) $\Gamma \vDash S^* \rightarrow S_2 +\!\!+ s' :: S$. By Lemma 6 we know that there is some S_3 such that $|S_3| > 0$ and $S^* = S_3 +\!\!+ S$. Since $|S_3| > 0$ we in particular know that $S_3 = S_3' +\!\!+ [s^3]$ for some S_3' and s^3. We do a case distinction on $|S_3'|$.

   a) Let $|S_3'| = 0$. We do another case distinction on the execution step performed in (A2)

      - In case of a local execution step we know that s^3 is a regular execution state so we can apply the inductive hypothesis on (A1) what gives us $s.\iota = s^3.\iota$. Further we have in this case that $S_2 = \epsilon$ and either s' is a terminal state or clearly $s'.\iota = s^3.\iota$ since local execution states do not affect the execution environment.
      - In case of the initiation of an internal transaction we have that s^3 is a regular execution state and $s' = s^3$ and $S_2 = [s^4]$ for some s^4. Hence we can apply the inductive hypothesis on (A1) what gives us $s.\iota = s^3.\iota$. and concludes the case.
      - The case of completing an internal transaction does not need to be considered since this would alter the substack S.

   b) If $|S_3'| > 0$ then we know by Lemma 17 that s^3 is a call state and hence a regular execution state. Hence we can apply the inductive hypothesis on (A1) yielding that $s.\iota = s^3.\iota$. We do another case distinction on the execution step performed in (A2)

      - If it is a local execution step, or the initiation of an internal transaction, or the completion of an internal transaction and $|S_3'| > 1$, then we have that $s' = s^3$ and hence the claim immediately follows.
      - If an internal transaction is completed and $S_3' = [s^4]$ for some s^4 then we know that $S_2 = \epsilon$. Futher from Lemma 17 we know that s^3 is a call state and hence also a regular execution state, so we can apply the inductive hypothesis that give us $s.\iota = s^3.\iota$. Further we know that $s'.\iota = s^3.\iota$ since applying return effects does not alter the execution environment.

□

Note that as the execution environment also the address of the annotation is persistent. This is not an immediate consequence of the persistence of the execution environment since also terminal execution states carry annotations.

Lemma 23 (Annotation Address Persistence)**.** *Let* $\Gamma \vDash S_1 +\!\!+ {s_{c_1}} :: S \ \rightarrow^n \ S_2 +\!\!+ {s'}_{c_2} :: S$. *Then it holds that address* $(c_1) =$ *address* (c_2).

Proof. If s and s' are regular execution state then the claim immediately follows from Lemma 22 since *address* $(c_1) = s.\iota.\mathsf{actor}$ and *address* $(c_2) = s'.\iota.\mathsf{actor}$. We make a case distinction:

1. If s is no regular execution state then $S_1 = \epsilon$ (due to Lemma 17). Further we know that $n = 0$ since otherwise we would have that (A1) $\Gamma \vDash s_{c_1} :: S \ \rightarrow \ S^*$ and (A2) $\Gamma \vDash S^* \ \rightarrow^{n-1} \ S_2 +\!\!+ {s'}_{c_2} :: S$. However if s is a terminal state, then the step in (A2)

would complete an internal transaction and hence $S = {s^1}_{c'} :: S'$ for some ${s^1}_{c'}$ and S' and $S^* = {s^2}_{c'} :: S'$ for some $s^2 \neq s^1$. However by Lemma 6 we know that $S^* = S_3 +\!\!+ S$. This clearly yields a contradiction since $S^* = S_3 +\!\!+ S = S_3 +\!\!+ {s^1}_{c'} :: S' \neq {s^2}_{c'} :: S' = S^*$.

2. If s is a regular execution state and s' is no regular execution state then we know that $n > 0$ (since $s \neq s'$) and (due to Lemma 17) that $S_2 = \epsilon$. Further one of the following would need to hold

   a) There must have been a regular execution state s^1 such that $\Gamma \vDash {s^1}_{c_2} :: S \rightarrow {s'}_{c_2} :: S$. Then by Lemma 22 we can show the claim since we know that $\mathit{address}\,(c_1) = s.\iota.\mathsf{actor}$ and $\mathit{address}\,(c_2) = s^1.\iota.\mathsf{actor}$.

   b) $\Gamma \vDash {s_{c_1}} :: S \rightarrow^{n-1} S$ and $\Gamma \vDash S \rightarrow {s'}_{c_2} :: S$ and $s' = \mathit{EXC}$. In this case however by 6 we know that $S = S_3 +\!\!+ S$ for some S_3 such that $|S_3| > 0$ what is clearly contradictory.

□

Even though the active account cannot be changed within the course of the execution, in case of a hash collision, the contract annotation might differ in this code. However, in the absence of hash collisions for the annotated contract, this is not possible:

Lemma 24 (Annotation Persistence)**.** *Let* $\Gamma \vDash S_1 +\!\!+ {s_{c_1}} :: S \rightarrow^* S_2 +\!\!+ {s'}_{c_2} :: S$ *be a collision-free execution of* $\mathcal{A}_T$ *and* $\mathit{address}\,(c_1) \in \mathcal{A}_T$*. Then it holds that* $c_1 = c_2$*.*

Proof. If s and s' are regular execution states then the claim immediately follows from Lemma 22 and Definition 20 since $\mathit{address}\,(c_1) = s.\iota.\mathsf{actor}$ and $\mathit{address}\,(c_2) = s'.\iota.\mathsf{actor}$, and $\mathit{code}\,(c_1) = s.\sigma(\mathit{address}\,(c_1)).\mathsf{code}$ and $\mathit{code}\,(c_2) = s'.\sigma(\mathit{address}\,(c_2)).\mathsf{code}$. So by Lemma 22 we have that $s.\iota.\mathsf{actor} = s'.\iota.\mathsf{actor}$ and by Definition 20 we know that $s.\sigma(\mathit{address}\,(c_1)).\mathsf{code} = s'.\sigma(\mathit{address}\,(c_1)).\mathsf{code}$. We make a case distinction for the other shapes of s and s':

1. If s is no regular execution state then $S_1 = \epsilon$ (due to Lemma 17). Further we know that $n = 0$ since otherwise we would have that (A1) $\Gamma \vDash {s_{c_1}} :: S \rightarrow S^*$ and (A2) $\Gamma \vDash S^* \rightarrow^{n-1} S_2 +\!\!+ {s'}_{c_2} :: S$. However if s is a terminal state, then the step in (A2) would complete an internal transaction and hence $S = {s^1}_{c'} :: S'$ for some ${s^1}_{c'}$ and S' and $S^* = {s^2}_{c'} :: S'$ for some $s^2 \neq s^1$. However by Lemma 6 we know that $S^* = S_3 +\!\!+ S$. This clearly yields a contradiction since $S^* = S_3 +\!\!+ S = S_3 +\!\!+ {s^1}_{c'} :: S' \neq {s^2}_{c'} :: S' = S^*$. Given that $n = 0$ we have ${s_{c_1}} = {s'}_{c_1}$ what proves the claim.

2. If s is a regular execution state and s' is no regular execution state then we know that $n > 0$ (since $s \neq s'$) and (due to Lemma 17) that $S_2 = \epsilon$. Further one of the following would need to hold

a) There must have been a regular execution state s^1 such that $\Gamma \vDash {s^1}_{c_2} :: S \rightarrow {s'}_{c_2} :: S$. Then by Lemma 22 and Definition 20 since $\mathit{address}\,(c_1) = s.\iota.\mathsf{actor}$ and $\mathit{address}\,(c_2) = s^1.\iota.\mathsf{actor}$, and $\mathit{code}\,(c_1) = s.\sigma(\mathit{address}\,(c_1)).\mathsf{code}$ and $\mathit{code}\,(c_2) = s^1.\sigma(\mathit{address}\,(c_2)).\mathsf{code}$. So by Lemma 22 we have that $s.\iota.\mathsf{actor} = s^1.\iota.\mathsf{actor}$ and by Definition 20 we know that $s.\sigma(\mathit{address}\,(c_1)).\mathsf{code} = s^1.\sigma(\mathit{address}\,(c_1)).\mathsf{code}$.

b) $\Gamma \vDash s_{c_1} :: S \rightarrow^{n-1} S$ and $\Gamma \vDash S \rightarrow {s'}_{c_2} :: S$ and $s' = \mathit{EXC}$. In this case however by 6 we know that $S = S_3 +\!\!+ S$ for some S_3 such that $|S_3| > 0$ what is clearly contradictory.

□

In order to justify the decision to take states that are consistent with their annotation c as an over-approximation of reachable states, we observe that each reachable execution state satisfies these properties.

Lemma 25 (Annotation Consistency of Reachable States)**.** *Let* $(\Gamma, s_c :: S)$ *be a reachable configuration. Then* s *is consistent with* c*.*

Proof. If $(\Gamma, s_c :: S)$ is reachable then there exists an initial state s^i such that $(\Gamma, {s^i}_{c_i}) = \mathit{initializeT}(T, H, \sigma, g)$ for some T, H, σ and g, and $\Gamma \vDash [{s^i}_{c_i}] \rightarrow^* s_c :: S$. By the definition of *initializeT* we know that s^i is consistent with c_i and hence by Lemma 21 we know that s is consistent with c. □

A.7 Proof Technique for Call Integrity

In this section, we formalize the correctness proof for the proof strategy for call integrity Theorem 1.

For proving that the three properties are sufficient for call integrity, we will need to show a strengthened version of the original statement.

For formulating this statement, we will first need to define some basic notions.

Definition 23 (Local Execution Step)**.** *We say that an execution step* $\Gamma \vDash S \rightarrow S'$ *is local if* $S = {s^1}_c :: S^*$ *and* $S' = {s^2}_c :: S^*$ *for some execution states* ${s^1}_c$, ${s^2}_c$ *and some call stack* S^**. We write*

$$\Gamma \vDash S \rightarrow_\ell S'$$

to denote a local execution step.

Lemma 26 (Preservation of Contract Codes by Local Execution)**.** *Let* $\Gamma \vDash s :: S \rightarrow_\ell s' :: S$*. Then* $\mathsf{codes}_{\mathcal{A}}(s) = \mathsf{codes}_{\mathcal{A}}(s')$

Proof. By an exhaustive case analysis on the small-step relation. □

Note that in particular this implies:

Lemma 27 (Preservation of Contract Codes by Local Multi-step Execution)**.** *Let* $\Gamma \vDash s :: S \rightarrow_\ell^n s' :: S$. *Then* $\mathsf{codes}_{\mathcal{A}}(s) = \mathsf{codes}_{\mathcal{A}}(s')$

Proof. By induction on n.

1. If $n = 0$ then $s = s'$ and the claim trivially holds.
2. If $n > 0$ then Let $\Gamma \vDash s :: S \rightarrow_\ell^{n-1} s'' :: S.$ and $\Gamma \vDash s'' :: S \rightarrow_\ell s' :: S.$ So the claim follows from Lemma 26 and the inductive hypothesis.

□

For formulating the strengthened statement, we will define what it means for to execution state of a contract c to agree on their call behavior with respect to c.

Definition 24 (Execution State Agreement on c-Call Behavior)**.** *We say that two execution states* s, t *(that are consistent with* c*) have the same call behavior w.r.t* c *(written* $s \sim_{\mathsf{calls}_c} t$*) if the following holds for all* Γ, S*:*

$$\begin{aligned}&\Gamma \vDash s_c :: S \xrightarrow{\pi}{}^* {s^f}_c :: S \ \wedge \ \mathit{final}\,(s^f)\\ &\Rightarrow \Gamma \vDash t_c :: S \xrightarrow{\rho}{}^* {t^f}_c :: S \ \wedge \ \mathit{final}\,(t^f)\\ &\Rightarrow \pi \downarrow_{\mathsf{calls}_c} = \rho \downarrow_{\mathsf{calls}_c}\end{aligned}$$

In the following we will write $\pi =_{\mathsf{calls}_c} \rho$ for $\pi \downarrow_{\mathsf{calls}_c} = \rho \downarrow_{\mathsf{calls}_c}$.

We introduce some other notions

- For two execution states s, t being equal up to the contract codes in some set of addresses $\mathcal{A}_C$ we write $s =_{\mathcal{A}_C - \mathsf{codes}} t$ which is formally defined as

$$\begin{aligned}&s = \mathit{EXC} \ \wedge t = \mathit{EXC}\\ &\vee \ s = \mathit{HALT}(\sigma, g, d, \eta) \ \wedge \ t = \mathit{HALT}(\sigma', d, g, \eta) \ \wedge \ \sigma =_{\mathcal{A}_C - \mathsf{codes}} \sigma'\\ &\vee \ s = (\mu, \iota, \sigma, \eta) \ \wedge \ t = (\mu, \iota, \sigma', \eta) \ \wedge \ \sigma =_{\mathcal{A}_C - \mathsf{codes}} \sigma'\end{aligned}$$

  where $\sigma =_{\mathcal{A}_C - \mathsf{codes}} \sigma'$ is defined as follows:

$$\begin{aligned}&(\forall a.\ a \notin \mathcal{A}_C \Rightarrow \sigma(a) = \sigma'(a))\\ &\wedge \ (\forall a.\ a \in \mathcal{A}_C \Rightarrow \sigma(a).\mathsf{b} = \sigma'(a).\mathsf{b} \wedge \sigma(a).\mathsf{stor} = \sigma'(a).\mathsf{stor} \wedge \sigma(a).\mathsf{n} = \sigma'(a).\mathsf{n})\end{aligned}$$

Code Independence We spell out in more detail the $\mathcal{A}_T$-code independence again. Technically, we do not want to only allow for different *static* code updates, but also for such that change over time. Consider e.g. the case where a contract reads the size of the code of an untrusted contract a_1, performs a call to another untrusted contract a_2 and then afterward compares the previously read code size of a_1 with the new one. Now usually, the code sizes should be the same. However, there is the possibility that new contracts are created and hence change their code. So if now a_2 actually created the contract a_1, then contract a_2 could influence the call behavior of contract c by creating contracts with different codes. This attack, however, is not relevant in practice since the creator of a contract cannot choose the address at which the contract will be created. Still, we should account for this scenario. By allowing for different code updates in the different local execution, we capture this case.

We in the following characterize small step execution under local code updates. We will, to this end, assume fs to be an infinite sequence of local code updates of the form $\mathsf{f} \in \mathcal{A} \to [\mathbb{B}^8]$. Further we will denote by $\Gamma \vDash s :: S \xrightarrow[\mathsf{f}\ \ell]{\pi}{}^{n} s^f :: S$ a local execution using the local code update f when accessing (other) contract codes. Note that the only way for a local execution to access the codes of other contracts is to use EXTCODESIZE and EXTCODECOPY, so only for these opcodes, the local code update will be used. Further, note that these opcodes, in the case, that the contract whose code is accessed does not exist return the same result as if the contract would have the empty code. Hence it is easy to locally mimic with a code update the original execution behavior.

$$\frac{\Gamma \vDash s :: S \xrightarrow[\mathsf{f}\ \ell]{\pi}{}^{n} s^f :: S}{\Gamma \vDash s :: S \xrightarrow[\mathsf{f::fs}]{\pi}{}^{n} s^f :: S} \qquad \frac{\Gamma \vDash s :: S \xrightarrow[\mathsf{f}\ \ell]{\pi_1}{}^{n} s^1 :: S \quad \Gamma \vDash s^1 :: S \xrightarrow{a} s^2 :: s^1 :: S\Gamma \vDash s^2 :: s^1 :: S \xrightarrow{\pi_2}{}^{m} S' +\!\!+ s^1 :: S}{\Gamma \vDash s :: S \xrightarrow[\pi^1 \cdot a \cdot \pi_2]{\mathsf{f::fs}}{}^{n+m+1} S' +\!\!+ s^1 :: S}$$

$$\frac{\begin{array}{c}\Gamma \vDash s :: S \xrightarrow[\mathsf{f}\ \ell]{\pi_1}{}^{n} s^1 :: S \\ \Gamma \vDash s^1 :: S \xrightarrow{a} s^2 :: s^1 :: S \qquad \Gamma \vDash s^2 :: s^1 :: S \xrightarrow{\pi_2}{}^{m} s^3 :: s^1 :: S \\ \Gamma \vDash s^3 :: s^1 :: S \to s^4 :: S \qquad \Gamma \vDash s^4 :: S \xrightarrow[\mathsf{fs}]{\pi_3}{}^{k} S' +\!\!+ S\end{array}}{\Gamma \vDash s :: S \xrightarrow[\mathsf{f::fs}]{\pi_1 \cdot a \cdot \pi_2 \cdot \pi_3}{}^{n+m+k+2} S' +\!\!+ S}$$

In particular, since the contract code in the global state is invariant over local executions, we can from each execution state s derive a local update sequence fs_s such that

$$\Gamma \vDash s :: S \xrightarrow{\pi}{}^{n} S' +\!\!+ S \Leftrightarrow \Gamma \vDash s :: S \xrightarrow[\mathsf{fs}_s]{\pi}{}^{n} S' +\!\!+ S$$

Effect Independence To reason more smoothly about effect independence, we define the notion of return update. Intuitively a return update describes the effects that the completion of an internal transaction has on the callee state.

Definition 25 (Return Update)**.** *A function $r_t \times \mathcal{S} \to \mathcal{S}$ is called the return update induced by t if the following condition holds:*

$$r_t(s) = s' \Leftrightarrow \exists \Gamma S.\ \Gamma \vDash t :: s :: S \to s' :: S$$

Note that return updates are well-defined since the small step relation is functional on calls tacks and indifferent towards the underlying call stacks (see Lemma 14). Further transaction completing execution steps are also indifferent towards transaction environments.

Further we introduce the following definition:

Definition 26 (Code Update)**.** *We define the substitution of the codes in execution state s by those in t (written $s[\mathsf{codes}(t)]$) as follows:*

$$s[\mathsf{codes}(t)] := \begin{cases} EXC & s = EXC \\ s & t = EXC \\ s[\sigma \to s.\sigma\ \langle a \to s.\sigma(a)[\mathsf{code} \to t.\sigma(a).\mathsf{code}]\rangle_{a \in \mathcal{A}}] & otherwise \end{cases}$$

We now can use these notions to give an alternative characterization of $\mathcal{A}_T$-effect independence

Definition 27 ($\mathcal{A}_T$-effect Independence)**.** *A contract $c \in \mathcal{C}$ is $\mathcal{A}_T$-effect independent of for a set of addresses $\mathcal{A}_T \subseteq \mathcal{A}$ if for all reachable configurations $(\Gamma, s_c :: S)$ such that $\Gamma \vDash s_c :: S \to s''_{c'} :: s_c :: S$ for some s'' and address $(c') \in \mathcal{A}_T$, it holds that for all final states s', t' with $\mathsf{codes}_{\mathcal{A}}(s) = \mathsf{codes}_{\mathcal{A}}(s') = \mathsf{codes}_{\mathcal{A}}(t')$ that*

$$r_{s'}(s) \sim_{\mathsf{calls}_c} r_{t'}(s)$$

Single-entrancy Single-entrancy gives that a reentering execution cannot perform another call. In particular, this implies that the trace of an internal transaction execution initiated by an execution of c cannot contain any call actions.

We capture this property in the following lemma:

Lemma 28. *Let (Γ, s_c) be a reachable configuration and let c be single-entrant. Further let $\Gamma \vDash s_c :: S \to^* {s^1}_{c'} :: s_c :: S$ and $\Gamma \vDash {s^1}_{c'} :: s_c :: S \xrightarrow{\pi}{}^* {s^2}_{c'} :: s_c :: S$. Then $\pi \downarrow_{\mathsf{calls}_c} = \epsilon$.*

Proof. Assume towards contradiction that $\pi \downarrow_{\mathsf{calls}_c} = a \cdot \pi_{call}$ for some call action a and call trace π_{call}. Then there must have been some execution such that $\Gamma \vDash {s^1}_{c'} :: s_c :: S \xrightarrow{\pi_1}{}^* S'$ and $\pi_1 \downarrow_{\mathsf{calls}_c} = \epsilon$ and $\Gamma \vDash S' \xrightarrow{a} S''$, and $\Gamma \vDash S' \xrightarrow{\pi_2}{}^* {s^2}_{c'} :: s_c :: S$, and $\pi_1 \downarrow_{\mathsf{calls}_c} = \pi_{call}$ By the definition of the small-step rules for transaction initiating instructions we know that $S'' = s^3_{c''} :: S'$. Further by Lemma 6 we know that there needs to be some S_3 such that $S' = S_3 +\!\!+ s_c :: S$ and $|S_3| > 0$. So consequently $S_3 = s_{3c'''} :: S'_3$ for some $s_{3c'''}$ and S'_3. Since a is a call action of contract c we can conclude that $c''' = c$. This immediately contradicts the single-entrancy of c given that $(\Gamma, s_c :: S)$ is reachable. □

Trusted Contracts We define all independence notions with respect to a set $\mathcal{A}_T$ of trusted contract addresses. For call integrity to hold, we need to assume that these contracts do not depend on non-trusted contracts themselves. Otherwise, call integrity could be trivially broken by, e.g., calling a trusted contract c_T that is performing some computations based on an untrusted contract c_U. If then contract c shows different behavior depending on the result of c_T this would break call integrity even though c itself would satisfy $\mathcal{A}_T$-effect independence (since it would only depend on the effects of a trusted contract code).

We hence define an integrity notion for trusted contracts that we will assume in the call integrity proof.

Definition 28 (Result Integrity)**.** *A contract $c \in \mathcal{C}$ satisfies result integrity for a set of addresses $\mathcal{A}_T \subseteq \mathcal{A}$ if for all reachable configurations $(\Gamma, s_c :: S), (\Gamma, s'_c :: S')$ with $s =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} s'$, it holds that for all t, t'*

$$\Gamma \vDash s_c :: S \xrightarrow{\pi}{}^{*} t_c :: S \ \wedge \ \mathit{final}\,(t_c) \ \wedge \ \Gamma \vDash s'_c :: S' \xrightarrow{\pi'}{}^{*} t'_c :: S' \ \wedge \ \mathit{final}\,(t'_c) \implies t =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} t'$$

In the following, we will assume all small step executions to be collision-free for the set of trusted contracts and the contract c for which the corresponding security notion is defined.

We finally prove the following lemma that implies call integrity

Lemma 29 (Call Behavior Agreement Invariance)**.** *Let c be a contract and let $\mathcal{A}_T$ be a set of addresses. Let (Γ, s_c) and (Γ, t_c) be reachable configurations and $s \sim_{\mathsf{calls}_c} t$. Further let $(\Gamma, \hat{s}_c)$ and $\hat{t}_c$ be reachable configurations such that $\hat{s} =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} s$ and $\hat{t} =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} t$. Further let c be single-entrant, $\mathcal{A}_T$-effect independent, and $\mathcal{A}_T$-code independent. Further let all $c_T \in \{(a, s.\sigma(a).\mathsf{code}) \mid a \in \mathcal{A}_T\}$ satisfy result integrity. Then also $\hat{s} \sim_{\mathsf{calls}_c} \hat{t}$.*

Proof. Let (A1) $s \sim_{\mathsf{calls}_c} t$, (A2) $\hat{s} =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} s$, (A3) $\hat{t} =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} t$, (A4) c be single-entrant, (A5) c be $\mathcal{A}_T$-effect independent, and (A6) c be $\mathcal{A}_T$-code independent, and (A7) all $c_T \in \{(a, s.\sigma(a).\mathsf{code}) \mid a \in \mathcal{A}_T\}$ satisfy result integrity. To show that $\hat{s} \sim_{\mathsf{calls}_c} \hat{t}$ by Definition 24, we assume that (B1) $\Gamma \vDash \hat{s}_c :: S \xrightarrow{\hat{\pi}}{}^{*} \hat{s}^f_c :: S$, (B2) $\mathit{final}\,(\hat{s}^f)$, (B3) $\Gamma \vDash \hat{t}_c :: S \xrightarrow{\hat{\rho}}{}^{*} \hat{t}^f_c :: S$, and (B4) $\mathit{final}\,(\hat{t}^f)$. And show that $\hat{pi} =_{\mathsf{calls}_c} \hat{\rho}$. We proceed by induction on $\hat{\pi} \downarrow_{\mathsf{calls}_c}$.

- Let $\hat{\pi} \downarrow_{\mathsf{calls}_c} = \epsilon$. Then the small step execution in (B1) did not perform any call, and so it coincides with a local execution: $\Gamma \vDash \hat{s}_c :: S \xrightarrow{\hat{\pi}}{}^{*}_{\ell} \hat{s}^f_c :: S$. Since $\hat{s} =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} s$, we know that in a local execution (which only accesses the account codes in the global state for the usage of EXTCODESIZE and EXTCODECOPY) that $\Gamma \vDash s_c :: S \xrightarrow[\mathsf{f}_{\hat{s}}]{\hat{\pi}}{}^{*}_{\ell} \tilde{s}^f_c :: S$ where $\mathsf{f}_{\hat{s}}$ denotes the code mapping induced by $\hat{s}$ and $\tilde{s}^f =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \hat{s}^f$. By Lemma 12 we know that $\Gamma \vDash s_c :: S \xrightarrow{\pi}{}^{*} s^f{}_c :: S$ for some s^f such that $\mathit{final}\,(s^f)$. By (A6) we know that $\hat{\pi} \downarrow_{\mathsf{calls}_c} = \pi \downarrow_{\mathsf{calls}_c} = \epsilon$. Further we know by Lemma 12 that $\Gamma \vDash t_c :: S \xrightarrow{\rho}{}^{*} t^f{}_c :: S$ and $\mathit{final}\,(t^f)$ for some t^f. Hence by (A1), we get that $\hat{\pi} \downarrow_{\mathsf{calls}_c} = \rho \downarrow_{\mathsf{calls}_c} = \epsilon$. Consequently the

execution of t does not perform any calls and corresponds to a local execution: $\Gamma \vDash t_c :: S \xrightarrow{\rho}{}^*_\ell\ t^f{}_c :: S$. Due to (A3) we know that this execution corresponds to $\Gamma \vDash \hat{t}_c :: S \xrightarrow[\rho]{\mathsf{f}_t}{}^*_\ell\ \tilde{t}^f_c :: S$ and $t =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \tilde{t}$, so also $\mathit{final}\,(\tilde{t})$. Then by (A6) and (B3) and (B4) we know that $\hat{\rho} \downarrow_{\mathsf{calls}_c} = \hat{\rho} \downarrow_{\mathsf{calls}_c} = \epsilon$ what concludes the case.

- Let $\hat{\pi} \downarrow_{\mathsf{calls}_c} = \hat{a} \cdot \hat{\pi}_{\mathit{call}}$. Then we know that the call $\hat{a}$ is the first to be performed by $\hat{s}$, so in particular, we have (C1) $\Gamma \vDash \hat{s}_c :: S \xrightarrow{\hat{\pi}_1}{}^*_\ell\ \hat{s}^1_c :: S$, (C2) $\Gamma \vDash \hat{s}^1_c :: S \xrightarrow{\hat{a}} \hat{s}^2_{c'} :: \hat{s}^1_c :: S$, (C3)$\Gamma \vDash \hat{s}^2_{c'} :: \hat{s}^1_c :: S \xrightarrow{\hat{\pi}_2}{}^*\ \hat{s}^3_{c'} :: \hat{s}^1_c :: S$, (C4)$\Gamma \vDash \hat{s}^3_{c'} :: \hat{s}^1_c :: S \xrightarrow{\epsilon} \hat{s}^4_c :: S$, and (C5) $\Gamma \vDash \hat{s}^4_c :: S \xrightarrow{\hat{\pi}_3}{}^*_\ell\ \hat{s}^f_c :: S$, and (C6) $\hat{\pi}_{\mathit{call}} = (\hat{\pi}_2 \cdot \hat{\pi}_3) \downarrow_{\mathsf{calls}_c}$. Due to single-entrancy of c we know by Lemma 28 that $\hat{\pi_2} \downarrow_{\mathsf{calls}_c} = \epsilon$, since otherwise a reentering execution of c would have initiated another internal transaction. Following the reasoning of the base case, we know from (C1) that (D1) $\Gamma \vDash s_c :: S \xrightarrow[\mathsf{f}_{\hat{s}}]{\hat{\pi}_1}{}^*_\ell\ \tilde{s}^1_c :: S$ for some $\tilde{s}^1$ such that $\tilde{s}^1 =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \hat{s}^1$. For this reason, $\tilde{s}^1$ will perform the same call $\hat{a}$, albeit this call might result in calling another contract than it does for $\hat{s}^1$. So (D2) $\Gamma \vDash \tilde{s}^1_c :: S \xrightarrow{\hat{a}} \tilde{s}^2_{c''} :: \tilde{s}^1_c :: S$. Still, we know by Lemma 12 that this execution will eventually end up in a terminal state of the initiated internal transaction, so (D3) $\Gamma \vDash \tilde{s}^2_{c''} :: \tilde{s}^1_c :: S \xrightarrow{\tilde{\pi}_2}{}^*\ \tilde{s}^3_{c''} :: \tilde{s}^1_c :: S$ and then will execute further till reaching a final state of original the execution of c: (D4) $\Gamma \vDash \tilde{s}^3_{c''} :: \tilde{s}^1_c :: S \xrightarrow{\epsilon} \tilde{s}^4_c :: S$, and (D5) $\Gamma \vDash \tilde{s}^4_c :: S \xrightarrow{\tilde{\pi}_3}{}^*_\ell\ \tilde{s}^f_c :: S$, and (D6) $\mathit{final}\,(\tilde{s}^f)$. Again we know due to single-entrancy of c that $\tilde{\pi_2} \downarrow_{\mathsf{calls}_c} = \epsilon$. We use the effect-independence (A5) and the inductive hypothesis to show that $\tilde{s}^4 \sim_{\mathsf{calls}_c} \hat{s}^4$. To this end we make a case distinction on the called address. Since the call action $\hat{a}$ agrees in both executions we know that also the called address does. Let in the following be $\mathit{to} = \mathit{address}\,(c') = \mathit{address}\,(c'')$

  1. If $\mathit{to} \in \mathcal{A}_T$ then we know that $c' = (\mathit{to}, \hat{s}.\sigma(\mathit{to}).\mathsf{code})$ and $c'' = (\mathit{to}, s.\sigma(\mathit{to}).\mathsf{code})$ and hence since $\hat{s} =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} s$ we have that $c' = c''$ and consequently also $\hat{s}^2 =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} s^2$ (since both execution states are initialized with the same call data). By (A7) we know that c' (and c'') satisfy result integrity and hence we get that $\hat{s}^3 =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} s^3$. So consequently also $\hat{s}^4 =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} s^4$ and so by the inductive hypothesis (since trivially $s^4 \sim_{\mathsf{calls}_c} s^4$ and $s^4 =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} s^4$) also $s^4 \sim_{\mathsf{calls}_c} \hat{s}^4$.
  2. If $\mathit{to} \notin \mathcal{A}_T$ then by the small step rules we know that $\hat{s}^4 = \mathsf{r}_{\hat{s}^3}(\hat{s}^1)$. Further, since we assume a collision-free execution we know that $\mathsf{codes}_{\mathcal{A}_T}(\hat{s}^3) = \mathsf{codes}_{\mathcal{A}_T}(\hat{s}^1)$ and hence $\mathsf{r}_{\hat{s}^3}(\hat{s}^1) =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \mathsf{r}_{\hat{s}^3[\mathsf{codes}(\hat{s}^1)]}(\hat{s}^1)$. Now by (A5) we can conclude that $\mathsf{r}_{\hat{s}^3[\mathsf{codes}(\hat{s}^1)]}(\hat{s}^1) \sim_{\mathsf{calls}_c} \mathsf{r}_{\tilde{s}^3[\mathsf{codes}(\hat{s}^1)]}(\tilde{s}^1)$. Now since $\hat{s}^1 =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \tilde{s}^1$ we know that $\mathsf{r}_{\tilde{s}^3[\mathsf{codes}(\hat{s}^1)]}(\hat{s}^1) =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \mathsf{r}_{\tilde{s}^3[\mathsf{codes}(\hat{s}^1)]}(\tilde{s}^1)$ (since either $\tilde{s}^3 = \mathit{EXC}$ in which case the states stay unchanged or their code will be changed to the codes in $\hat{s}^1$). And now since $\mathsf{codes}_{\mathcal{A}_T}(\hat{s}^1) = \mathsf{codes}_{\mathcal{A}_T}(\tilde{s}^1)$ (since $\hat{s}^1 =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \tilde{s}^1$) and $\mathsf{codes}_{\mathcal{A}_T}(\tilde{s}^3) = \mathsf{codes}_{\mathcal{A}_T}(\tilde{s}^1)$ (since we assume collission resistant executions), we also know that $\mathsf{r}_{\tilde{s}^3[\mathsf{codes}(\hat{s}^1)]}(\tilde{s}^1) =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \mathsf{r}_{\tilde{s}^3[\mathsf{codes}(\tilde{s}^3)]}(\tilde{s}^1) = \tilde{s}^4$. By the inductive hypothesis we can hence conclude that $\tilde{s}^4 \sim_{\mathsf{calls}_c} \hat{s}^4$ and consequently $\hat{\pi}_3 \downarrow_{\mathsf{calls}_c} = \tilde{\pi}_3 \downarrow_{\mathsf{calls}_c}$.

  Next we consider a (terminal) execution of s. Let (E) $\Gamma \vDash s_c :: S \xrightarrow{\pi}{}^*\ s^f{}_c :: S$ be the terminal

execution from s. Then we know that this corresponds to the execution under local updates described by the update sequence fs_s induced by s: $\Gamma \vDash s_c :: S \xrightarrow[\mathsf{fs}_s]{\pi}{}^{*} s^f{}_c :: S$. Further we know that $\Gamma \vDash s_c :: S \xrightarrow[\mathsf{f}_{\hat{s}}::\mathsf{fs}_{\tilde{s}4}]{\tilde{\pi}}{}^{*} s^f{}_c :: S$ where $\tilde{\pi} = \hat{\pi}_1 \cdot \hat{a} \cdot \tilde{\pi}_2 \cdot \tilde{\pi}_3$. Hence by (A6) we can conclude that (E0) $\pi \downarrow_{\mathsf{calls}_c} = \tilde{\pi} \downarrow_{\mathsf{calls}_c}$ and hence also the first call action performed by s in the terminal execution is $\hat{a}$. Consequently we have that (E1) $\Gamma \vDash s_c :: S \xrightarrow{\pi_1}{}^{*}_{\ell} s^1{}_c :: S$ for some s^1 such that $\mathsf{codes}_{\mathcal{A}}(s^1) = \mathsf{codes}_{\mathcal{A}}(s)$ (since a local execution does not effect the codes of other contracts, Lemma 27), (E2) $\Gamma \vDash s^1{}_c :: S \xrightarrow{\hat{a}} s^2{}_{c''} :: s^1{}_c :: S$. Still, we know by Lemma 12 that this execution will eventually end up in a terminal state of the initiated internal transaction, so (E3) $\Gamma \vDash s^2{}_{c''} :: s^1{}_c :: S \xrightarrow{\pi_2}{}^{*} s^3{}_{c''} :: s^1{}_c :: S$ and then will execute further till reaching a final state of original the execution of c: (E4) $\Gamma \vDash s^3{}_{c''} :: s^1{}_c :: S \xrightarrow{\epsilon} s^4{}_c :: S$, and (E5) $\Gamma \vDash s^4{}_c :: S \xrightarrow{\pi_3}{}^{*}_{\ell} s^f{}_c :: S$, and (E6) $\mathit{final}\,(s^f)$, and (E7) $\pi = \pi_1 \cdot \hat{a} \cdot \pi_2 \cdot \pi_3$. Due to single-entrancy we can again conclude that $\pi_2 \downarrow_{\mathsf{calls}_c} = \epsilon$, and so by (E0) that $\pi_3 \downarrow_{\mathsf{calls}_c} = \tilde{\pi}_3 \downarrow_{\mathsf{calls}_c}$. Next, we consider a (terminal execution) of t: Let (F) $\Gamma \vDash t_c :: S \xrightarrow{\rho}{}^{*} t^f{}_c :: S$ be the terminal execution from t. Then by (A1) we know that (F0) $\pi \downarrow_{\mathsf{calls}_c} = \rho \downarrow_{\mathsf{calls}_c}$. Hence, in particular, we know that $\hat{a}$ is the first call action performed by the execution of t and hence we have: (F1) $\Gamma \vDash t_c :: S \xrightarrow{\rho_1}{}^{*}_{\ell} t^1{}_c :: S$ for some t^1 such that $\mathsf{codes}_{\mathcal{A}}(t^1) = \mathsf{codes}_{\mathcal{A}}(t)$ (since a local execution does not effect the codes of other contracts, Lemma 27), (F2) $\Gamma \vDash t^1{}_c :: S \xrightarrow{\hat{a}} t^2{}_{c''} :: t^1{}_c :: S$. Still, we know by Lemma 12 that this execution will eventually end up in a terminal state of the initiated internal transaction, so (F3) $\Gamma \vDash t^2{}_{c''} :: t^1{}_c :: S \xrightarrow{\rho_2}{}^{*} t^3{}_{c''} :: t^1{}_c :: S$ and then will execute further till reaching a final state of original the execution of c: (F4) $\Gamma \vDash t^3{}_{c''} :: t^1{}_c :: S \xrightarrow{\epsilon} t^4{}_c :: S$, and (F5) $\Gamma \vDash t^4{}_c :: S \xrightarrow{\rho_3}{}^{*}_{\ell} t^f{}_c :: S$, and (F6) $\mathit{final}\,(t^f)$, and (F7) $\pi = \rho_1 \cdot \hat{a} \cdot \rho_2 \cdot \rho_3$. Due to single-entrancy we can again conclude that $\rho_2 \downarrow_{\mathsf{calls}_c} = \epsilon$, and so by (F0) that $\rho_3 \downarrow_{\mathsf{calls}_c} = \pi_3 \downarrow_{\mathsf{calls}_c}$. Following the reasoning of the base case, we know from (F1) that (G1) $\Gamma \vDash \hat{t}_c :: S \xrightarrow[\mathsf{f}_t \ \ell]{\rho_1}{}^{*} \tilde{t}^1_c :: S$ for some $\tilde{t}^1$ such that $\tilde{t}^1 =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} t^1$. For this reason $\tilde{t}^1$ will perform the same call $\hat{a}$ as t^1, albeit this call might result in calling another contract than it does for t^1. So (G2) $\Gamma \vDash \tilde{t}^1_c :: S \xrightarrow{\hat{a}} \tilde{t}^2_{c'''} :: \tilde{t}^1_c :: S$. Still, we know by Lemma 12 that this execution will eventually end up in a terminal state of the initiated internal transaction, so (G3) $\Gamma \vDash \tilde{t}^2_{c'''} :: \tilde{t}^1_c :: S \xrightarrow{\tilde{\rho}_2}{}^{*} \tilde{t}^3_{c'''} :: \tilde{t}^1_c :: S$ and then will execute further till reaching a final state of original the execution of c: (G4) $\Gamma \vDash \tilde{t}^3_{c'''} :: \tilde{t}^1_c :: S \xrightarrow{\epsilon} \tilde{t}^4_c :: S$, and (G5) $\Gamma \vDash \tilde{t}^4_c :: S \xrightarrow{\tilde{\rho}_3}{}^{*}_{\ell} \tilde{t}^f_c :: S$, and (G6) $\mathit{final}\,(\tilde{t}^f)$. Again we know due to single-entrancy of c that $\tilde{\rho}_2 \downarrow_{\mathsf{calls}_c} = \epsilon$. We use the effect-independence (A5) and the inductive hypothesis to show that $\tilde{t}^4 \sim_{\mathsf{calls}_c} t^4$. To this end we make a case distinction on the called address. Since the call action $\hat{a}$ agrees in both executions we know that also the called address does. Let in the following be $\mathit{to} = \mathit{address}\,(c'') = \mathit{address}\,(c''')$

1. If $\mathit{to} \in \mathcal{A}_T$ then we know that $c'' = (\mathit{to}, t.\sigma(\mathit{to}).\mathsf{code})$ and $c'' = (\mathit{to}, \hat{t}.\sigma(\mathit{to}).\mathsf{code})$ and hence since $s =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \hat{t}$ we have that $c'' = c'''$ and consequently also $t^2 =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \tilde{t}^2$ (since both execution states are initialized with the same data call data). By (A7) we know that c'' (and c''') satisfy result integrity and hence we get that $t^3 =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \tilde{t}^3$. So consequently also $t^4 =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \tilde{t}^4$ and so by the inductive hypothesis (since trivially

$\tilde{t}^4 \sim_{\mathsf{calls}_c} \tilde{t}^4$ and $\tilde{t}^4 =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \tilde{t}^4$) also $\tilde{t}^4 \sim_{\mathsf{calls}_c} t^4$.

2. If *to* $\notin \mathcal{A}_T$ then by the small step rules we know that $t^4 = \mathsf{r}_{t^3}(t^1)$. Further, since we assume a collision-free execution we know that $\mathsf{codes}_{\mathcal{A}_T}(t^3) = \mathsf{codes}_{\mathcal{A}_T}(t^1)$ and hence $\mathsf{r}_{t^3}(t^1) =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \mathsf{r}_{t^3[\mathsf{codes}(t^1)]}(t^1)$. Now by (A5) we can conclude that $\mathsf{r}_{t^3[\mathsf{codes}(t^1)]}(t^1) \sim_{\mathsf{calls}_c} \mathsf{r}_{\tilde{t}^3[\mathsf{codes}(t^1)]}(t^1)$. Now since $t^1 =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \tilde{t}^1$ we know that $\mathsf{r}_{\tilde{t}^3[\mathsf{codes}(t^1)]}(t^1) =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \mathsf{r}_{\tilde{t}^3[\mathsf{codes}(t^1)]}(\tilde{t}^1)$ (since either $\tilde{t}^3 = \mathit{EXC}$ in which case the caller states stay unchanged or their code will be changed to the codes in t^1). And now since $\mathsf{codes}_{\mathcal{A}_T}(t^1) = \mathsf{codes}_{\mathcal{A}_T}(\tilde{t}^1)$ (since $t^1 =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \tilde{t}^1$) and $\mathsf{codes}_{\mathcal{A}_T}(\tilde{t}^3) = \mathsf{codes}_{\mathcal{A}_T}(\tilde{t}^1)$ (since we assume collission resistant executions), we also know that $\mathsf{r}_{\tilde{t}^3[\mathsf{codes}(t^1)]}(\tilde{t}^1) =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} \mathsf{r}_{\tilde{t}^3[\mathsf{codes}(\tilde{t}^3)]}(\tilde{t}^1) = \tilde{t}^4$. By the inductive hypothesis we can hence conclude that $\tilde{t}^4 \sim_{\mathsf{calls}_c} t^4$ and consequently $\rho_3 \downarrow_{\mathsf{calls}_c} = \tilde{\rho}_3 \downarrow_{\mathsf{calls}_c}$.

Next we consider the (terminal) execution of $\hat{t}$ given in (B3). Then we know that this corresponds to the execution under local updates described by the update sequence $\mathsf{fs}_{\hat{t}}$ induced by $\hat{t}$: $\Gamma \vDash \hat{t}_c :: S \xrightarrow[\mathsf{fs}_{\hat{t}}]{\hat{\rho}}{}^{*} \hat{t}_c^f :: S$. Further we know that $\Gamma \vDash \hat{t}_c :: S \xrightarrow[\mathsf{f}_t::\mathsf{fs}_{\tilde{t}^4}]{\tilde{\rho}}{}^{*} \hat{t}_c^f :: S$ where $\tilde{\rho} = \rho_1 \cdot \hat{a} \cdot \tilde{\rho}_2 \cdot \tilde{\rho}_3$. Hence by (A6) we can conclude that (H0) $\hat{\rho} \downarrow_{\mathsf{calls}_c} = \tilde{\rho} \downarrow_{\mathsf{calls}_c}$. This concludes the proof since

$$\begin{aligned}
\hat{\pi} \downarrow_{\mathsf{calls}_c} &= \hat{a} \cdot (\hat{\pi}_3 \downarrow_{\mathsf{calls}_c}) \\
&= \hat{a} \cdot (\tilde{\pi}_3 \downarrow_{\mathsf{calls}_c}) = \tilde{\pi} \downarrow_{\mathsf{calls}_c} \\
&= \hat{a} \cdot (\pi_3 \downarrow_{\mathsf{calls}_c}) = \pi \downarrow_{\mathsf{calls}_c} \\
&= \hat{a} \cdot (\rho_3 \downarrow_{\mathsf{calls}_c}) = \rho \downarrow_{\mathsf{calls}_c} \\
&= \hat{a} \cdot (\tilde{\rho}_3 \downarrow_{\mathsf{calls}_c}) = \tilde{\rho} \downarrow_{\mathsf{calls}_c} \\
&= \hat{\rho} \downarrow_{\mathsf{calls}_c}
\end{aligned}$$

□

This lemma trivially implies the correctness of the proof strategy for call integrity:

Theorem 4 (Proof Strategy for Call Integrity)**.** *Let $c \in \mathcal{C}$ be a contract and let (Γ, s_c) and (Γ, t_c) be reachable configurations of c such that $s =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} t$. Further let $\mathcal{A}_T \subseteq \mathcal{A}$ a subset of trusted addresses such that $c_T \in \{(a, s.\sigma(a).\mathsf{code}) \mid a \in \mathcal{A}_T\}$. If c is $\mathcal{A}_T$-code independent, c is $\mathcal{A}_T$-effect independent, and c is single-entrant then it holds that $s \sim_{\mathsf{calls}_c} t$.*

Proof. It trivially holds that $s =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} s \sim_{\mathsf{calls}_c} s$. So since $s =_{\mathcal{A}/\mathcal{A}_T-\mathsf{codes}} t$ we can immediately apply Lemma 29 to obtain $s \sim_{\mathsf{calls}_c} t$. □

APPENDIX B

Appendix to Chapter 3

B.1 Soundness Issues in Related Work

We review the soundness problems of other works on automated static smart contract analysis. We thereby focus on those works that make soundness claims. We first overview soundness problems in the reconstruction of smart contracts' control flow graphs (which particularly affects the Securify analyzer [TDDC+18]) and afterwards successively discuss the issues in the analyses performed by [TDDC+18], [LWZ+19], [GMS18a], and [KGDS18a]. Where possible, we provide reproducible evidence in form of concrete counter-examples for the spotted sources of unsoundness.

B.1.1 Control Flow Reconstruction

Most of the tools that analyze Ethereum smart contracts at the level of bytecode base their analysis on the contract's control flow graph (CFG). However, the design of the EVM bytecode language does not allow for an easy reconstruction of a contract's control flow since jump destinations are not statically fixed but might be dynamically computed. More precisely, in EVM, bytecode jump destinations are read from the stack and hence can be subject to prior computations. Even though the set of potential jump destinations is statically determined (since only such program counters with a JUMPDEST instruction constitute valid jump destinations), the concrete destination of a jump instruction might only be dispatched at runtime. The challenge hence lies in statically narrowing down the set of possible jump destinations for each branch instruction (JUMP or JUMPI). To this end, the state-of-the-art analyzer [TDDC+18] employs a custom algorithm, another popular solution [oB18] uses an external open-source tool [cfg20] for control flow graph reconstruction. While correctness for both of them has never been discussed, flaws in the CFG reconstruction can lead to catastrophic consequences: An unsound reconstruction that erroneously excludes possible jump destinations can deem parts of the contract code unreachable that carries critical and potentially unsafe functionality (e.g., reentrant calls).

When reviewing the algorithms used in [TDDC+18] and [cfg20], we found soundness issues in both approaches, as we will discuss in the following. In Figure B.1 we show a compact example of a smart contract's control flow that is recovered incorrectly by [TDDC+18, cfg20] with no errors reported. Intuitively, the control flow of this contract should not be fully recoverable

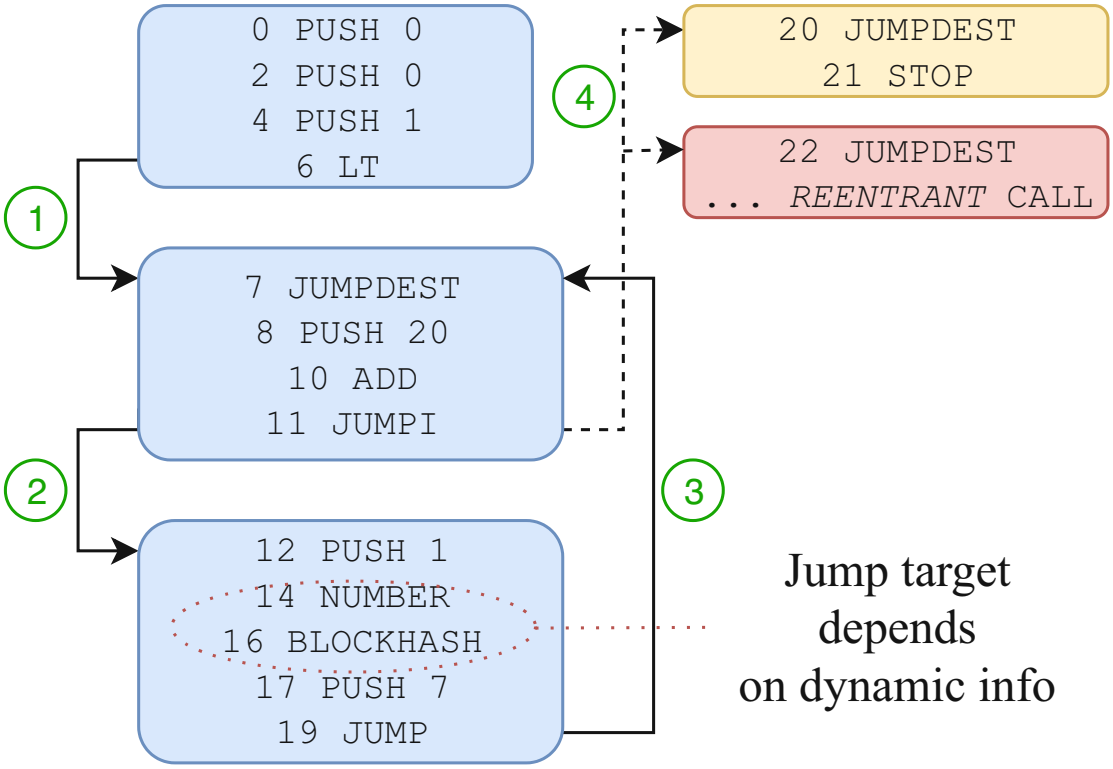

Figure B.1: Problematic Control Flow Example.

because one of its jump destinations depends on some blockchain information (the block hash and the block number) which cannot be statically predicted, but will only be fixed once the contract has been published on the blockchain.

The smart contract is structured into five basic blocks. The first block (starting at program counter 0), initializes the local machine check with two 0 values and continues with the execution of the second block starting at program counter 7 (①). The second block can be entered via a jump (since it starts with a JUMPDEST instruction). It intuitively takes two stack values as arguments, the first one functioning as jump offset and the second being the jump condition: it computes the next jump destination as the sum of 20 and the top stack element and conditionally jumps to this destination based on the second stack value. In the first iteration, since both of these values are 0 (and so particularly the condition is 0), no jump is performed, but instead, the execution proceeds with block three (starting at program counter 12) with the empty stack (②). This block pushes the current block number and hash to the stack and jumps back to the second block (③). Since, at this point, the input to the second block are values that are not statically determinable, it

needs to be assumed that the jump condition, as well as the jump offset, could have any value. It is hence possible during the real execution to jump to arbitrary jump destinations from program counter 10 (④). This includes the block starting at program counter 20 where the execution of the contract is stopped, and most importantly, the block starting at program counter 22 that executes a reentrant call. Thus, if this jump destination is undiscovered, false correctness results for reentrancy can be produced in subsequent analysis.

There are two sound approaches for handling the usage of unpredictable information in jump destination reconstruction: Conservatively, a smart contract can be rejected by the analysis and hence be considered potentially vulnerable in this case (which is our approach) or the analysis could assume that all JUMPDEST instructions of the contract are potentially reachable. The tools that we reviewed, however, did not follow any of these options but produced the following results: [cfg20] correctly discovers the basic blocks, but cannot recover jumps to the targets 20 and 22 (④). The result of [TDDC+18] is even more surprising: the algorithm does not manage to recover any of the blocks shown in Figure B.1, but reports as CFG of this contract a single block consisting of a modulo instruction followed by the STOP opcode. Consequently, all analyses that use either of these CFG reconstruction solutions will consider the reentrant call of the example contract to be unreachable and will, based on that, label the contract as safe to reentrancy attacks.

B.1.2 Securify

The Securify tool [TDDC+18] encodes dependencies inferred from a contract's control flow graph as logical facts and specifies security properties in terms of compliance and violation patterns using these facts. It is claimed that the satisfaction of a compliance pattern is sufficient for proving a security property, while matching a violation pattern guarantees that a security property is indeed violated. We will, in the following, review most of the provided patterns and give counterexamples, showing that most of these patterns indeed are not sound. We validated as far as possible the patterns reported in the paper with the provided online tool (`https://securify.chainsecurity.com`)[1] . Unfortunately, some of the patterns introduced in [TDDC+18] were changed or renamed in the online tool. We will note this when discussing the corresponding pattern. Also, it should be noted that the online tool only reports security problems. More precisely, an alarm (red) is produced if a violation pattern is matched, a warning (orange) is produced if neither a violation nor a compliance pattern is matched. The lack of a report for a certain security property indicates that the property's compliance pattern was matched.

Ether Liquidity. The LQ (Ether liquidity) property ensures that a property cannot lock Ether (for this reason, it is called Locked Ether in the online tool), meaning that for all the contract's executions either leave the contract's balance unaffected or there is a trace that allows to reduce the contract's balance.

The property formulates three different compliance patterns. The first two compliance patterns ensure that all halting instructions are preceded by a successful conditional check on the value given to the call being 0. This ensures that only such executions can complete successfully,

[1] We accessed the website on January 19th and validated all properties with *Solidity* Compiler version 0.4.25.

which have been guaranteed to have gotten no money transferred. These patterns are probably sufficient to guarantee that a contract can never receive money and hence for showing the LQ property. The third compliance pattern checks whether there is a call that is reachable while at the same time (meaning that the call and an exception opcode are not reachable from the same conditional branch) no exception is reachable, and this call transfers a non-zero value or a value that is settable by the environment. This shall ensure that the contract has at least one way of successfully transferring money.

This compliance pattern is not sufficient for ensuring Ether Liquidity. Despite the problem that the corresponding Ether transferring call could be restricted to a certain address which can never initiate such a call (as it belongs to a contract without functionality to call other contracts), the pattern also does not consider that an exception that reverts the transaction might not only occur conditionally.

Consider the following contract:

```
1 contract Bob {
2   function sendMoney(address c) {
3     c.send(2);
4     throw;
5   }
6   function receive() payable {
7   }
8 }
```

This contract is labeled not to lock Ether even though it can receive money (via the `receive` function), and every Ether transfer to another contract (via `sendMoney`) will be reverted. Note that the absence of *Solidity*'s `payable` will be translated to a conditional check on the call value, and cause a revert once if the value given to the call is non-zero.

The violation pattern for LQ requires that there is no `CALL` instruction that transfers a non-zero amount of Ether and that there is some halting instruction such that if its reachability is dependent on a conditional branching, this condition can be determined by the transaction data, hence can be enabled by the transaction initiator. This shall ensure that there is at least one execution trace that does not halt exceptionally and hence reverts the execution effect.

However, the following contract is reported to lock Ether (matches the violation pattern):

```
1 contract Bob {
2   function receive(uint x) payable {
3     if (x > 0 || x <= 0) {
4       throw;
5     }
6   }
7 }
```

This contract clearly cannot lock Ether since it cannot receive any Ether. Its only function `receive` throws an exception depending on a conditional, which is always true. Still, the dependency analysis labels this condition to be determined by transaction data so that the pattern is matched. This can also be considered as a soundness flaw in Securify's definition of

determinability since, in this case, clearly, for different values of transaction data, the value of the condition is the same (which contradicts [TDDC+18]'s definition of determinability.)

No writes after calls. The NW (No writes after calls) property says that a contract's storage when terminating the execution should always be the same as at the point of a previous contract call (so the contract shall not be altered between a CALL instruction and the contract's successful termination). The online tool does not implement a property with such a name but instead implements similar patterns for a property called Gas-dependent Reentrancy. This property uses the same intuition but puts an additional requirement that the amount of gas given to the call shall depend on the remaining gas. One should note that it is very misleading that this property is in the online tool called Gas-dependent Reentrancy, even though [TDDC+18] explicitly claims that the NW property is different from reentrancy. We will detail out later why the NW property indeed is not a sound or complete approximation of the single-entrancy property.

The corresponding compliance pattern requires that the CALL instruction may not be followed by an SSTORE instruction. This pattern, however, does not consider that there are other ways of modifying the storage than the SSTORE instruction, e.g., by using DELEGATECALL for calling a library function that alters the storage.

Consider the following example:

```
1 library Lib {
2   struct Data { bool bvalue; }
3   function write(Data storage self, bool value) {
4     self.bvalue = value;
5   }
6 }
7
8 contract Bob {
9   Lib.Data sent;
10
11   constructor () {
12     sent.bvalue = false;
13    }
14
15   function ping(address c) {
16     if (!(sent.bvalue)) {
17       if (!c.call.value(2)()) {
18         throw;
19       }
20       Lib.write(sent, true);
21     }
22   }
23 }
```

This example clearly matches the compliance pattern (since the call to the library will be translated to a DELEGATECALL instruction, hence no SSTORE instruction appears in the first place). This can also be verified with the online tool, which does not report a violation of the Gas Dependent Reentrancy property.

The violation pattern for the NW property requires that there is a CALL that must be proceeded by an SSTORE instruction. This pattern is also not sufficient as illustrated by the following example:

```
contract Alice {
  bool sent = false;

  function ping(address c) {
    if (!sent) {
      sent = true;
      c.call.value(2)();
      sent = sent;
    }
  }
}
```

This contract clearly does not violate the property (since the contract storage at the point of terminating is not altered as compared to the point of calling). Still, it matches the violation pattern (and is reported by the online tool), indicating a guaranteed property violation.

Next, we shortly discuss why the NW property (independently of the fact that the patterns are not sufficient) is neither sound nor complete for single-entrancy.

We will first give an example of a contract satisfying the NW property while still being reentrant.

```
contract Bank {
  address a; address b;
  uint balA; uint balB;

  function setBalA (uint v) {
    balA = v;
  }

  function drainA(address ben) {
    if (msg.sender != a) { throw; }
    if (balA > 0) {
      uint v = balA;
      setBalA(0);
      ben.call.value(v)();
    }
  }
}
```

This contract implementing a simple bank functionality for two parties (identified by their addresses `a` and `b`) is vulnerable to a reentrancy attack even though no writes after the call are performed. Similar to the initial example in Figure 3.3, given that `a` is the address of a malicious contract, this contract can use the public `setBalA` function in a reentering execution to disable the guard (here `balA`) before reentering the contract's `drainA` function to retransfer money that `a` does not own.

For an example of a contract that does not satisfy the NW property but that is still safe, we give a contract with a simple locking functionality (similar to the example in Figure 3.3).

```
contract Bank {
  uint lock;
  mapping (address => uint) bal;

  function drain(address a) {
    if (lock == 1) { throw; }
```

```
7     lock = 1;
8     a.call.value(bal[msg.sender])();
9     bal[msg.sender] = 0;
10    lock = 0;
11  }
12 }
```

The locking ensures that whenever the function is reentered, an exception occurs, and hence no further call can be performed. Still, since the `lock` needs to be released at the end of the execution. Clearly, the NW property is violated.

Restricted write. The RW (restricted write) property requires that all write accesses are restricted, meaning there is at least one address that, when initiating the call, cannot reach the corresponding write access.

This property needs to be questioned in its semantic definition in that this definition explicitly requires that SSTORE instructions are not reachable even though (as discussed before), the SSTORE instruction is not the only way of manipulating storage.

So, for example, when analyzing the following contract, there is no RW violation or warning produced for the `Bob` contract even though the `ping` functions allows to set contract's `data` filed containing the owner to be set to an arbitrary value by anyone.

```
1 library Lib {
2   struct Data { address owner; }
3   function write(Data storage self, address value) {
4     self.owner = value;
5   }
6 }
7
8 contract Bob {
9   Lib.Data data;
10
11  function ping(address c) {
12    Lib.write(data, c);
13  }
14 }
```

The given compliance pattern requires that the storage offset specified in a SSTORE instruction needs to be determined by the caller of the contract. This pattern might indeed be sufficient for the semantic property only considering SSTORE instructions.

The violation pattern requires that the reachability of SSTORE instructions as well as the offset given to them may not depend on the caller of the contract.

However, in the following contract an unrestricted write is detected:

```
1 contract Test {
2   bool test = false;
3
4   function flipper () {
5     if (msg.sender != 0)
6       flip();
7   }
8   function flip () internal {
```

```
9     test = !test;
10  }
11 }
```

This contract should be safe with respect to the semantic definition since `flip`, the only function containing write access is an internal function, meaning that it can only be invoked within the contract. Given that the only place where it is invoked (in the `flipper` function), this is done with a restriction on the caller (`msg.sender`), also this storage access is restricted. However, the contract is reported to match the violation pattern. A reason for that could be an unsoundness in the underlying dependency analysis.

Restricted transfer. The RT (restricted transfer property) excludes that Ether transfers (via CALL) cannot be invoked by any user. Again one could criticize that the property does not consider other ways of transferring money (e.g., by CALLCODE). The following contract, for example, is considered safe by this definition:

```
1 contract Bob {
2   function sendMoney(address c) {
3     c.callcode.value(5)();
4   }
5 }
```

The corresponding compliance pattern requires that all calls transfer 0 Ether. Given that the property only considers CALL instructions, this pattern is probably sufficient.

There are two violation patterns for the RT property; the first one requires that there is a CALL instruction transferring a non-zero amount and whose reachability may be dependent on the caller. We can give a counterexample similar to the one for the RW violation pattern:

```
1 contract Test {
2   function sendMoney() {
3     if (msg.sender != 1)
4       sendM();
5   }
6   function sendM () internal {
7     msg.sender.send(1);
8   }
9 }
```

Again, the tool does not detect that effectively the money transfer is restricted since the internal function `sendM` can only be invoked in a restricted fashion.

The second violation pattern for the RT property requires instead of the transferred value to be non-zero that the value is determined by the input to the call while at the same point the input might not affect the reachability of the CALL instruction.

We can again give a simple counterexample similar to the previous one:

```
1 contract Test {
2   function sendMoney(uint x) {
3     if (msg.sender != 1)
4       sendM(x);
5   }
```

```
6   function sendM (uint y) internal {
7     msg.sender.send(y);
8   }
9 }
```

This example is detected as insecure while having only restricted money transfers.

Handled exception. The HE property (Handled exception) is not semantically defined but intuitively shall ensure that exceptions that occurred in function calls shall be handled. Due to the lack of a formal definition, it is hard to argue to which extend the given patterns really are sufficient, but we give here examples of proper/problematic exception handling, which are wrongly classified.

The compliance pattern requires that every call must be followed by some branching instruction whose condition is determined by the call's return value. Clearly, the following contract is matched by this pattern even though it does not perform a proper exception handling.

```
1 contract SimpleBank {
2   mapping(address => uint) balances;
3   uint successes;
4
5   function withdraw() {
6     bool success = msg.sender.send(balances[msg.sender]);
7     if (success) { successes++; }
8     balances[msg.sender] = 0;
9   }
10 }
```

Even though this contract branches on the return value of the call, this branching does not influence the critical instruction, namely the following storage update that assumes a successful call.

The violation pattern for HE requires that all branching instructions following a CALL instruction do not have a condition that depends on the outcome of the call. We give an example of a contract matching this pattern that however implements a useful form of exception handling:

```
1 library Lib {
2   function toInt(bool b) returns (uint n) {
3     if (b)
4       return 1;
5     else
6       return 0;
7   }
8 }
9
10 contract SimpleBank {
11   mapping(address => uint) balances;
12
13   function withdraw() {
14     bool success = msg.sender.send(balances[msg.sender]);
15     balances[msg.sender] = Lib.toInt(success) * balances[msg.sender];
16   }
17 }
```

This contract uses the return value of the call to update the callee's balance after the call depending on that. Since the branching on the return value is outsourced to the library function `toInt`, it can not be captured by the corresponding pattern.

In general, it is hard to imagine how proper exception handling should be generically defined since this is a property that depends in the end on the contract's desired functionality.

Transaction ordering dependency. The TOD (Transaction ordering dependency) property is again not formally defined but requires that the order of other transactions shall not influence the calls of the contract. More precisely, calls shall not depend on a state that can be altered by other transactions. The paper says that actually different types of dependency will be considered distinguishing whether the amount to be transferred (TA), the receiver (TR), or the reachability of the CALL as a whole are affected (TT). However, it seems that TT is not implemented since not even the following straight forward TT violating contract is detected by the online tool:

```
1 contract SimpleGame {
2   uint counter = 0;
3
4   function play() {
5     counter = 10;
6   }
7
8   function getReward() {
9     if (counter > 0) {
10      msg.sender.send(10);
11    }
12  }
13 }
```

The compliance pattern for TOD requires that calls shall not depend on the contract's storage or balance. Again this property does not consider that there are different ways of calling, e.g., using CALLCODE.

The following contract is considered secure:

```
1 contract Bob {
2   uint price;
3
4   function setPrice(uint v) {
5     price = v;
6   }
7
8   function sendMoney(address c) {
9     c.callcode.value(price)();
10  }
11 }
```

This contract transfers an amount of money (`price`) that it reads from the storage, and that could have been modified by another transaction before. Still, no warning about a TOD violation is triggered by the online tool.

The violation pattern requires that there is a CALL which depends on a read of a constant storage cell that can be written.

Consider the following example contract:

```
1 contract Bob {
2   uint price = 5;
3
4   function sendMoney(address c) {
5     price = price;
6     c.send(price);
7   }
8 }
```

This contract is labeled to be TOD even though the transferred amount is constantly 5 and cannot be influenced by any other transaction.

Validated arguments. The VA (Validated arguments) property is again not semantically specified but shall ensure that arguments to a function are checked for meeting desired preconditions. Similarly to the HE property, it is unclear how such a goal should be captured by a generic property.

The compliance pattern requires that such values that depend on input value may only be written to the global storage if they have previously been checked, meaning that must have been a conditional branching before whose condition depended on the argument.

The following contract is an easy example of a contract matching the compliance pattern while not performing proper argument validation:

```
1 contract Test {
2   uint test;
3   uint count = 0;
4
5   function setTest (uint x) {
6     if (x < 10) {
7       count++;
8     }
9     test = x;
10  }
11 }
```

Even though the write to the storage of argument variable `x` is preceded by a corresponding conditional branch, this check does not influence whether the variable is indeed written to storage.

The violation pattern for VA requires that there is a storage instruction writing a value dependent on an argument that is not preceded by a corresponding conditional branch with a condition dependent on the argument.

The following contract performs a proper argument validation but is still matched by the violation pattern.

```
1 library Lib {
2   function validateArgument(uint i) {
3     if (!(i >= 0 && i < 100))
4       throw;
5   }
6 }
7
```

```
8 contract Test {
9   uint test;
10
11   function setTest (uint x) {
12     Lib.validateArgument(x);
13     test = x;
14   }
15 }
```

Since the validation is performed by the library function `validateArgument`, the conditional branch which performs the validation cannot be detected.

B.1.3 NeuCheck

The tool NeuCheck[LWZ+19] analyses Ethereum Smart contracts written in the *Solidity* by checking the contract's syntax graph for specific patterns. The tool is claimed to be sound even though no concrete soundness claim is formulated.

The formulated patterns are purely syntactic and can be rather seen as a check for compliance with certain style guidelines. Take as an example the access control pattern: this pattern checks whether all functions have modifiers such as `private` or `internal` defined, which restrict general access. It is unclear which semantic property should be implied by this pattern. Clearly, there are safe usages of public functions as well as incorrect access control (e.g., due to a wrong party being allowed to call a certain contract function) even though a contract function is restricted by some modifier.

For illustrating the issues of this syntactic pattern-based approach further, we will, in the following, review the reentrancy pattern as this is particularly interesting for our case: The reentrancy pattern checks for the occurrences of *Solidity*'s `call` function that do not have a gas limit checks and checks whether this occurrence is followed by the assignment of a state variable. First, the absence of checking a gas limit does not ensure that reentrancy attacks are not possible. If not gas limit is set, the gas given to the call is computed with respect to the remaining gas. So one could easily set a high gas limit or even set all remaining gas of the execution as a gas limit (if the amount specified exceeds the remaining gas, the same gas as in the case of a lacking specification is given to the call). On top of this, as discussed for Securify, setting a variable assignment is not the only way of changing the state (this can also be done via a library call). The counterexample for Securify's NW compliance pattern would also be a valid counterexample for this case. Similarly, Securify's NW violation pattern would serve as a counterexample for the pattern's completeness. As a consequence, matching the reentrancy pattern clearly does not guarantee the absence of a reentrancy attack.

Unfortunately, we could not experimentally assess the unsoundness of the provided patterns since we did not find a way to build the tool from the provided sources[2]. The assessment of the tool is further aggravated by the fact that the paper gives the corresponding patterns in PseudoCode

[2]Sources are made available at `https://github.com/Northeastern-University-Blockchain/NeuCheck`. We contacted the authors at the end of November for clarification of the building process but received no reply as of January 20th, 2020.

that leaves many crucial details (in particular how the dependency structure between syntactic constructs is established) undefined.

B.1.4 EtherTrust

The approach to sound smart contract analysis presented in [GMS18a] exhibits an unsoundness when it comes to modeling reentering executions. More precisely, the proposed abstraction assumes that the contract's storage at the point of reentering is the same as at the point of calling. This is not necessarily the case since another (malicious) contract might, in the meanwhile, manipulate the storage of the corresponding contract by invoking other state-changing functions of it. An example is the `DAO` contract depicted in Fig. 3.3. This contract would be deemed secure according to the abstraction presented in [GMS18a] since the described attack requires to change the value of the contract's `lock` variable by an invocation of the `switchLock` function prior to reentering the `withdraw` function. If it is assumed that the storage at the point of reentering the `withdraw` function is the same as at the point of invoking the `call` method, the contract would be secure since the `lock` variable is always set to `1` when calling hence preventing to reach the `call` method when reentering.

B.1.5 ZEUS

A recently published work is the analysis tool ZEUS [KGDS18a] that analyses smart contracts written in *Solidity* using symbolic model checking. The analysis proceeds by translating *Solidity* code to an abstract intermediate language that again is translated to LLVM bitcode. Finally, existing symbolic model checking tools for LLVM bitcode are leveraged for performing the analysis. The security properties are defined in terms of XACML style policies that are translated to state reachability assertions in the intermediate language (and finally to assertions in LLVM bitcode). The authors evaluate their tool for generic security properties (such as reentrancy), which are not expressed in terms of policies (which are contract specific) but by an informal description of how to add specific assertions to contracts of interest. For some properties, e.g., reentrancy, the insertion of assertions is not sufficient, and additional program modifications need to be applied to the original contracts. The authors claim their tool to be sound which they support by a proof sketch and empirical results. This claim, however, has several shortcomings:

- There is no formal soundness statement made. In particular, there is no formal relation between the policy compliance of *Solidity* contracts and the analysis results established and also not covered in the proof sketch.
- The proof is sketchy and exhibits several holes and at least two flaws: While there is an intuitive argument why given the translation from *Solidity* to the abstract intermediate language are correct and adding assertions does not influence semantics, there is no proof provided for the statement that the translation from the intermediate language to LLVM bitcode preserves soundness. That this property does not hold is (indirectly) admitted by the authors as they discuss that the compiler optimizations on LLVM bitcode remove relevant contract behavior. Consequently, assuming that compiler optimizations on LLVM bitcode are semantics preserving,

this clearly contradicts that the translation from the intermediate language preserves semantics. For one particular optimization, a fix is hardcoded, but there is no formal argument given that this particular fix is sufficient for establishing soundness. Also the claim that the provided translation from *Solidity* to the intermediate language is faithful can be clearly contradicted. This is due to a clear deviation in the call semantics of the intermediate language from the *Solidity* semantics. The mechanism underlying *Solidity*'s call functionalities is the one of the CALL instructions in EVM bytecode. In particular, this mechanism determines that the failure of a contract call causes the revocation of the global state to the point of calling. The proposed semantics of the intermediate language, however does not allow for such a revocation (even by design). This is similar to the issue in the semantics used in Oyente that we described in Section 2.3.7.

- The final results for the predefined properties (such as reentrancy) are not covered by the soundness claim as there is no (formal) argument made that the performed program modifications are sound. In particular the presented method for detecting same-function reentrancies is faulty: For detecting same-function reentrancy of a function f, f is replicated (resulting in f') and the *Solidity*'s `call` construct in f is replaced by a call to f' whose occurrence of `call` is preceded by a false assertion for proving the unreachability of the corresponding call. This treatment is problematic in several ways: First, the use of the `call` construct is not the only way of calling another contract, indeed it is way more common to use direct calls. Second, similar to the problem discussed for [GMS18a], such an abstraction fails to detect the example of Figure 3.3, even though this is clearly a case of same-function reentrancy. The problem is that for the used approach of replacing calls by invocations to f' it is assumed that a call can at most be preceded by a direct invocation of f without any other state-changing function calls being happening in the meanwhile. Consequently, single-entrancy (and even same-function single-entrancy) is a property that cannot be assessed by considering certain contract parts in isolation. Consider the following to contracts:

```
1 contract Bank{
2   uint lock;
3   mapping (address => uint) bal;
4
5   function take () {
6     lock = 1;
7   }
8
9   function release () {
10    lock = 0;
11  }
12
13  function drain(address a) {
14    if (lock == 1) { throw; }
15      lock = 1;
16      a.call.value(bal[msg.sender])();
17      bal[msg.sender] = 0;
18      lock = 0;
19   }
20 }
```

```
1 contract Bank{
2   uint lock;
```

```
3   mapping (address => uint) bal;
4
5   function drain(address a) {
6     if (lock == 1) { throw; }
7       lock = 1;
8       a.call.value(bal[msg.sender])();
9       bal[msg.sender] = 0;
10      lock = 0;
11  }
12 }
```

Even though the implementation of the `drain` function is identical in both contracts, the first contract allows for a (same-function) reentrancy attack while the second does not. ZEUS, however, would label both of these contracts to be safe.

Unfortunately, we were not able to conduct an empirical evaluation of the described issues since no sources for ZEUS are made available. Our request to the authors of [KGDS18a] to provide us with sources or binaries that would allow us to experimentally access ZEUS has been denied. For this reason, we were forced to conduct our comparison with ZEUS on the publicly available dataset for which [KGDS18a] reports numbers. We further discuss this dataset in the following.

Problems in the ZEUS dataset. While comparing *HoRSt* against the dataset used in [KGDS18a][3] we encountered several problems. The dataset is a list of 1524 contracts with the classification provided by ZEUS and the assessment of whether the authors consider this classification correct. No source or bytecode is provided.

Of these 1524 contracts, 21 have a name that does not resemble a Ethereum address (e.g. `Code_3_fdf6d_faucet`). Of the remaining 1503, 397 actually have a truncated address (i.e., 39 instead of 40 hexadecimal digits). The remaining 1106 addresses contain duplicates. After removing them, we arrive at 1033 addresses. For 286 of these addresses, we were not able to obtain the bytecode: 53 have been self-destructed according to `https://etherscan.io` which makes retrieving their bytecode non-trivial, 232 have no recorded transaction (in particular no transaction that created them) and 1 is an external account (i.e., an address with no code deployed). This leaves us with 747 addresses. After removing contracts with the same bytecode, we arrive at 720 contracts[4]. We contacted the authors of [KGDS18a] on July 16th, 2019, about these problems and received no answer as of January 20th, 2020.

[3] `https://docs.google.com/spreadsheets/d/12_g-pKsCtp3lUmT2AXngsqkBGSEoE6xNH51e-of_Za8`

[4] Note that the authors of [KGDS18a] deduplicated their dataset on the source level, therefore it may well be that these same bytecodes were produced by different source codes

APPENDIX C

Appendix to Chapter 4

The appendix is structured as follows: In section C.1 we overview the analysis specification language *HoRSt*. In section C.2 we make the theoretical foundations of our work explicit, in particular we define its relation to abstract interpretation and give the soundness proof. In section C.3 gives details on how the security properties discussed in Chapter 4 are implemented in *eThor* using the specification language *HoRSt*.

C.1 *HoRSt*

This section gives an introduction to the newly developed language *HoRSt* that allows for the high-level specification of Horn-clause-based static analyses. We will first give a short primer that illustrates the main functionality of *HoRSt*, followed by a more detailed discussion of the optimizations performed by the *HoRSt* compiler.

C.1.1 *HoRSt* by Example

For illustrating the features of *HoRSt* we show how to express a general rule for binary stack operations, subsuming the rule for addition presented in Section 4.2. Figure C.1 shows an excerpt of the *HoRSt*-specification of the presented static analysis. The abstract domain of the analysis is realized by the definition of the abstract data type `AbsDom`. Predicate signatures can be specified by corresponding predicate declarations as done for the case of the `MState` predicate. *HoRSt* allows for parametrizing predicates and thereby specifying whole predicate families: The `MState` predicate is parametrized by two integer values (as specified in the curly braces) that will intuitively correspond to the contract's identifier and the program counter whose state it is approximating. The arguments of the `MState` predicate family reflect exactly those specified in section 4.2.

To facilitate modular specifications, *HoRSt* supports non-recursive operations over arbitrary types, such as `absadd` which implements abstract addition. In the example, we show the flexibility of

```
1 datatype AbsDom := @T | @V<int>; // Abstract Domain
2 datatype Opcode := @STOP | @ADD | ... | @INVALID | @SELFDESTRUCT // opcodes (
      shortened)
3
4 pred MState{int*int}: int * array<AbsDom> * array<AbsDom> * array<AbsDom> * bool;
5
6 op absadd(a: AbsDom, b: AbsDom): AbsDom := match (a, b) with | (@V(x), @V(y)) => @V
      ((x + y) mod MAX) | _ => @T;
7 op binOp(c: Opcode, x: AbsDom, y: AbsDom): AbsDom := match c with | @ADD => absadd(
      x, y) | ... | _ => @T;
8
9 sel ids: unit -> [int]; // contracts to be analyzed
10 sel binOps: unit -> [int]; // binary stack operations
11 sel pcsForIdAndOpcode: int * int -> [int]; // program counters at which a specific
      opcode occurs in a specific contract
12 sel argumentsTwoForIdAndPc: int * int -> [int * int]; // results from the
      preanalysis for a given contract and pc
13
14 op tryConcrete{!c:int}(val:AbsDom): AbsDom := (!c = ~1) ? (val) : (@V(!c));
15
16 rule opBin :=
17   for (!op: int) in binOps(), (!id: int) in ids(), (!pc: int) in pcsForIdAndOpcode
        (!id, !op),
18       (!a:int, !b: int) in argumentsTwoForIdAndPc(!id, !pc)
19    clause [?x: AbsDom, ?y:AbsDom, ?size: int, ?sa: array<AbsDom>, ?mem: array<
         AbsDom>, ?stor: array<AbsDom>, ?cl: bool]
20      MState{!id, !pc}(?size, ?sa, ?mem, ?stor, ?cl), ?size > 1,
21      ?x = tryConcrete{!a} (select ?sa (?size -1)), ?y = tryConcrete{!b} (select ?sa
            (?size -2))
22      => MState{!id, !pc +1}(?size -1, store ?sa (?size -2) (binOp(intToOpCode(!op),
            ?x,?y)), ?mem, ?stor, ?cl);
```

Figure C.1: *HoRSt* rule describing the abstract semantics of local binary stack operations.

HoRSt by presenting a single rule template for generating rules for all binary stack operations. To this end, we define a function `binOp` that, given an opcode `c` and two integer arguments, applies the binary operation corresponding to the opcode to the provided arguments. This function is then leveraged in the rule template `opBin`. Rule templates serve for generating the abstract semantics given in the form of Horn clauses. As in our case, the abstract semantics is specified as a function on a concrete contract, the generation of Horn clauses in *HoRSt* needs to be linked to a concrete contract bytecode. In order to account for that in a generic fashion, given that *HoRSt* cannot support facilities for reading files or parsing bytecodes, *HoRSt* provides an interface for interacting with custom relations generated by *Java*™ code. This interface is specified upfront by so-called *selector functions* (introduced with the keyword `sel`), which are declared, but not defined in the *HoRSt* specification. In the example, we declare selector functions for accessing the identifiers of the contracts to be analyzed (`ids`), the set of binary operations (`binOps`), and the program counters in a contract that hold opcodes of a specific type (`pcForIdAndOpcode`). In addition to that, selector functions also allow for more advanced functionalities such as incorporating the results of a preanalysis in an elegant fashion: To this end, we declare the selector function `argumentsTwoForIdAndPc` that returns arguments to the operation that could be statically precomputed (returning -1 in case of failure). For generating Horn clauses, we can parametrize

the rule over the cross product of the result of (nested) selector function applications as done in for the `opBin` rule. This then exactly generates Horn clauses abstracting the behavior of a binary stack operation as discussed in section 4.2: A stack size check is performed, the two arguments are selected from the stack, and finally, the `MState` predicate at the next program counter is implied with an updated stack having the operation's result as the top element. The only derivation occurs due to the consideration of the preanalysis: the operation `tryConcrete` tries to access the statically precomputed argument, and only in case of its absence performs the (more expensive) stack access. This step, however, is not a necessity but just illustrates how the interplay between different stages of a static analysis can be implemented for boosting the performance.

C.1.2 *HoRSt* in Detail

In the following, we present a short overview of the features of *HoRSt*.

Types and Operations. For specifying the super domain $\mathcal{D}$ of the abstraction, *HoRSt* provides in addition to the primitive types BOOLEAN and INTEGER, non-recursive sum types and arrays over all types. The type of abstract values $\hat{D}$ used in section 4.2 that consists of the unknown value $\top$ and concrete integer values, can be defined as follows:

```
1 datatype AbsDom = @T | @V<int>;
```

In addition, *HoRSt* allows us to define non-recursive operations over arbitrary types. These operations are implemented as hygienic macros on the expression level. To work with sum types, *HoRSt* provides match expressions. This mechanism can, e.g., be used to define the abstract addition operation described in Figure 4.2.2 as follows:

```
1 op absadd(a: AbsDom, b: AbsDom): AbsDom :=
2   match (a, b) with
3   | (@V(x), @V(y)) => @V(x + y) // for two concrete values, return sum
4   | _              => @T;       // else return top
```

Predicates. The abstraction's predicate signature $\mathcal{S}$ is given in terms of predicate declarations. A predicate declaration introduces a predicate symbol that ranges over arguments of arbitrary types. *HoRSt* supports a mechanism for declaring a whole family of predicates with the same argument types by allowing for the specification of compile-time constants that we will from now on call *parameters*. We illustrate the syntax of predicate declarations with the predicate $\mathsf{MState}_{\mathsf{pc}}$ defined in Figure 4.2 that models an abstract execution state:

```
1 pred MState{int}: int * array<AbsDom> * array<AbsDom> * array<AbsDom>  * bool;
```

The declared predicate has one parameter of type `int` and five arguments. The parameter represents the program counter pc and should be considered part of the predicate name. The distinction between parameters and arguments is supported by *HoRSt* for performance reasons: different parameter instantiations are compiled to different predicate names in the underlying SMT representation leading to speed-ups in practice and additionally facilitates the folding optimization discussed in section 4.3. **Selector Functions.** *HoRSt* itself provides no facilities to

read files, parse bytecode, etc. Instead, these tasks are handled by *Java*™ code. *HoRSt* interacts with this *Java*™ code by an upfront-specified interface which is implemented by so-called *selector functions*. The tasks performed by selector functions can be as easy as providing an integer interval or as complicated as precomputing the results of certain bytecode operations, from a *HoRSt* perspective, we only see the interface provided by *selector function declarations* that associate selector function names with their type signature. Selector functions are restricted to take a fixed number of arguments of primitive types and to return a sequence of tuples of primitive types.

Examples of selector function declarations are given below:

```
1 sel interval: int -> [int]; // integers from 0 to (n-1)
2 sel pcsForOpcode: int -> [int]; // program counters for given opcode
3 sel pcsAndValuesForOpcode: int -> [int*int]; // program counters and precomputed
      values
```

In general, selector functions can be seen as the bridge between the analysis specification and the parts of the software stack responsible for preprocessing (parsing, etc.) real-world smart contracts. For instance, as previously discussed, the predicate signature $\mathcal{S}_{c^*}$, the abstraction function α_{c^*} as well as the abstract semantics $\delta(c^*)$ are dependent on the concrete contract c^* under analysis. Selector functions allow us to implement such a parametrization (e.g., iterating over opcode sequences in order to generate rules in $\delta(c^*)$ according to the opcode at each program counter).

The separation of concerns introduced by selector functions helps to keep the *HoRSt* specifications declarative while the technical details of providing the actual values can be tested by unit tests.

Rules. The fundamental abstraction of *HoRSt* is the concept of *rule*, which essentially describes a collection of Horn clauses. It, therefore, can be seen as the mechanism for specifying the abstract semantics $\delta(c^*)$. A rule is either a singleton rule that is just instantiated once or may act as a template for arbitrarily many instantiations – hence describing a family of rules. The second case is enabled by the use of selector functions, which provide the sequence that the rule that family ranges over. More technically, for each tuple returned by a selector function, the parameters of the rule template will be instantiated according to the tuple values.

The rule shown in Figure C.2 for example will be instantiated for all program counters `!pc` at which c^* holds an MSTORE instruction. The sequence of these program counters is provided by the selector function `pcsForOpcode` that maps opcodes to their corresponding set of occurrences (identified by program counter) in c^*.

Within the body of rules, we can define (optionally hygienic) macros that we can use in the subsequent *clauses* of the rule. The clauses themselves (declared with keyword `clause`), describe a Horn clause consisting of a list of premises and a conclusion ranging over free variables which need to be explicitly declared upfront. Premises are lists of predicate applications and boolean *HoRSt* expressions, while the conclusion may only consist of a single predicate application. The example in Figure C.2 defines three clauses that exactly correspond to the Horn clauses defined for $(\!|\mathsf{MSTORE}|\!)_{\mathsf{pc}}$ in Figure 4.4.

```
1 op valToMemWord (v: AbsDom, mem: array<AbsDom>, o: int): array<AbsDom> :=
2   for (!a: int) in interval(32): x: array<AbsDom> -> store x (o + !a) (
        absExtractByteL{!a}(v)), mem;
3 op isConcrete(a: AbsDom): bool := match a with | @T => false | _ => true;
4 op extractConcrete(a: AbsDom): int := match a with | @V(x) => x | _ => 0;
5
6 rule opMstore :=
7   for (!id: int) in ids(),
8       (!pc:int) in pcsForIdAndOpcode(!id, MSTORE),
9       (!p: int, !v: int) in argumentsTwoForIdAndPc(!id, !pc)
10    clause [?size: int, ?sa: array<AbsDom>, ?mem: array<AbsDom>, ?stor: array<
          AbsDom>, ?cl: bool, ?offset: AbsDom, ?p: int, ?v: AbsDom]
11      MState{!id,!pc}(?size, ?sa, ?mem, ?stor, ?cl), ?size > 1,
12      !p != ~1,
13      ?v = tryConcrete{!v}(select ?sa (?size -2))
14      => MState{!id, !pc +1}(?size - 2, ?sa, writeWord{!p}(?v, ?mem), ?stor, ?cl),
15    clause [?size: int, ?sa: array<AbsDom>, ?mem: array<AbsDom>, ?stor: array<
          AbsDom>, ?cl: bool, ?pos: AbsDom,
16            ?v: AbsDom, ?memn: array<AbsDom>]
17      MState{!id,!pc}(?size, ?sa, ?mem, ?stor, ?cl), ?size > 1,
18      !p = ~1,
19      ?pos = select ?sa (?size -1),
20      ?v = tryConcrete{!v}(select ?sa (?size -2)),
21      ?memn = (isConcrete(?pos)) ? (writeWordEven(extractConcrete(?pos), ?v, ?mem))
              : ([@T])
22      => MState{!id, !pc +1}(?size - 2, ?sa, ?memn, ?stor, ?cl);
```

Figure C.2: *HoRSt* rule describing the abstract semantics of the local memory write operation $(\!|MSTORE|\!)_{\mathsf{pc}}$

Sum Expressions. Selector functions can not only be used to generate rules but can also be used at the expression level. So-called sum expressions exist in two different shapes: in the simple case (shown later in Figure C.10), predefined associative operations (addition, multiplication, disjunction, and conjunction) are used to join expressions that may make use of the values returned by the selector function; the generalized case can be seen in line 2 of Figure C.2. The operation `valToMemWord` updates 32 consecutive memory cells of `mem` with fractions of the value `v` starting from position `o` — `mem` is the start value, **`store`** `x (o + !a)( absExtractByteL{!a}(v))` is the iterated expression (`x` acts as a placeholder for the last iteration step's result).

Queries. In order to check for reachability of abstract configurations, *HoRSt* allows for the specification of (reachability) queries that can also be generated from selector functions. The query that is shown in Figure C.6 for instance, checks for reentrancy by checking if any CALL instruction is reachable with call level $c\ell = 1$ (here encoded as bool). It, therefore, is an implementation of the reachability property introduced in subsection 4.2.5.

Note that if there is a notion of an expected outcome, we can define queries with the keyword **`test`** as seen in Figure C.10.

```
query reentrancyCall
  for (!id: int) in ids(),
      (!pc:int) in pcsForIdAndOpcode(!id, CALL)
    [?sa: array<AbsDom>, ?mem: array<AbsDom>,
     ?stor: array<AbsDom>, ?size:int]
      MState{!id, !pc}(?size, ?sa, ?mem, ?stor, true);
```

Figure C.3: *HoRSt*-query for reeentrancy.

$$\left.\begin{array}{r} P_1(x) \wedge y = x + 1 \Rightarrow P_2(y) \\ P_2(y) \wedge z = y * 3 \Rightarrow P_3(z) \end{array}\right\} P_1(x) \wedge y = x + 1 \wedge z = y * 3 \Rightarrow P_3(z)$$

Figure C.4: Unfolding of P_2.

C.1.3 Compiler Optimizations

The *HoRSt* compiler features several transformations to generate optimized `smt-lib` output. It first resolves high-level language constructs such as operations and datatypes and unfolds the parametrization introduced by selector functions. The resulting basic Horn clause representation is optimized by constant folding, and optionally, by performing different flavors of the unfolding transformation to eliminate predicates that are not relevant for the queries. In the following, we explain the unfolding transformation in more detail.

The idea behind the unfolding transformation is that a predicate p can be eliminated from a set of Horn clauses Λ by unfolding the occurrences of p in the premises according to the clauses that have p as conclusion. An example is given in Figure C.4. Here predicate P_2 is eliminated by merging the two single execution steps (modeled by the two clauses on the left) into a combined clause (on the right) summarizing the steps.

This intuition serves as a starting point for the unfolding strategy of *linear folding*. In linear folding, all clauses representing a basic block of sequential execution steps are merged into a single clause. More precisely, the unfolding transformation is only applied to those predicates that are used linearly in Λ, meaning that p occurs in the premises of exactly one clause in Λ and in the conclusion of exactly one different clause in Λ. Linear folding has the advantage that it runs linearly in the number of clauses in Λ and, as a result, yields a reduced set of clauses Λ' such that $|\Lambda'| \leq |\Lambda|$.

In contrast, applying the unfolding transformation exhaustively on all predicates (except those that are recursively used) might yield an exponential blow-up in clauses (and hence also result in exponential runtime). In practice, however, the set of clauses Λ' resulting from such a *exhaustive folding* is often of a reasonable size. For mitigating the runtime overhead, however, it is crucial to avoid unnecessary blow-ups in the intermediate clause sets produced during the transformation: To this end, for exhaustive folding *HoRSt* applies linear folding first and only afterward performs the unfoldings that multiply existing clauses.

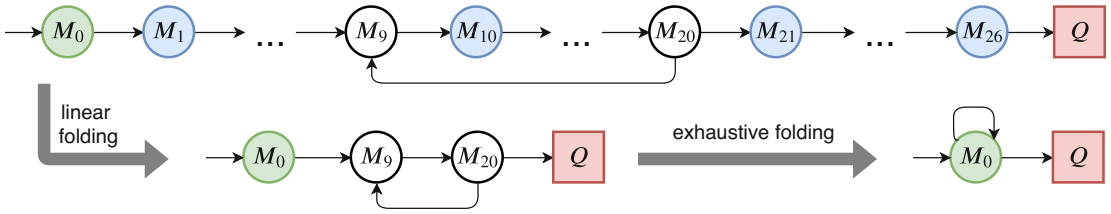

Figure C.5: Example of linear and exhaustive folding. Transition system view of the abstract semantics: States denote predicates and arrows denote Horn clauses having the start predicate as premise and the goal predicate as conclusion. Initial (final) states are colored green (red). Linearly used predicates are colored blue.

Figure C.5 shows the effects of linear and exhaustive folding for a simple contract with a loop. Since the abstraction at each program counter is modeled by an own predicate (MState, here M for short), the contract control flow is reflected by the logical dependencies between these predicates, as defined in the Horn clauses of the abstract semantics. Therefore, we depict the abstract semantics as a transition system, interpreting predicates as states and Horn clauses as transitions. Linear folding collapses all sequences with linear control flow, while exhaustive folding, in this case, reduces the state space even further without adding additional clauses.

C.2 Theoretical Foundations of *eThor*

In this section, we provide details on the theoretical foundations of *eThor*. We first formally characterize the notion of Horn-clause-based abstractions as they can be implemented in *HoRSt* and then relate this concept to the framework of abstract interpretation. Next, we provide missing details on the definition of the static analysis underlying *eThor* and conclude with the proof of the soundness statement for this analysis.

C.2.1 Horn-clause-based Abstractions

In this section, we more formally characterize the aim and scope of this work, as well as the kind of static analyses that are realizable by *HoRSt*. Generally, we focus on the reachability analysis of programs with a small-step semantics, which we over-approximate by an abstract program semantics based on Horn clauses. More formally, we will assume a program's small-step semantics to be a binary relation S_s over program configurations $c \in \mathcal{C}$. A Horn-clause-based abstraction for such a small-step semantics S_s is then fully specified by a tuple $(\mathcal{D}, \mathcal{S}, \alpha, \Lambda)$ where $\mathcal{S}$ defines the signature of predicates with arguments ranging over (partially) ordered subsets of $\mathcal{D}$. For a given predicate signature $\mathcal{S}$, an abstraction function $\alpha : \mathcal{C} \to \mathcal{A}$ maps concrete program configurations $c \in \mathcal{C}$ to abstract program configurations $\Delta \in \mathcal{A}$ consisting of instances of predicates in $\mathcal{S}$.

Formally, a predicate signature $\mathcal{S} \in \mathcal{N} \nrightarrow \prod(\mathcal{P}(\mathcal{D}) \times (\mathcal{P}(\mathcal{D}) \times \mathcal{P}(\mathcal{D})))$ is a partial function from predicate names $\mathcal{N}$ to their argument types (formally written as a product over the subsets

of some abstract superdomain $\mathcal{D}$, equipped with a corresponding order). We require for all $n \in \mathcal{N}$ that $(D, \leq) \in \mathcal{S}(n)$ such that $(D, \leq)$ forms a partially ordered set. Correspondingly, the set of abstract configurations $\mathcal{A}_\mathcal{S}$ over $\mathcal{S}$ can be defined as $\mathcal{P}(\{n(\vec{v}) \mid n \in \mathcal{N} \ \wedge \ \forall i \in \{1, \ldots, |\mathcal{S}(n)|\}. \pi_i(\mathcal{S}(n)) = (D, \leq) \implies \pi_i(\vec{v}) \in D\})$ where $\pi_i(\cdot)$ denotes the usual projection operator. The abstraction of a small-step semantics is then a set of constrained Horn clauses $\Lambda \subseteq \mathcal{H}(\mathcal{S})$ that approximates the small-step execution rules.

A constrained Horn clause is a first order formula of the form

$$\forall X.\ \Phi, P \Rightarrow c$$

Where $X \subseteq \mathit{Vars} \times \mathcal{P}(\mathcal{D})$ is a (functional) set of typed variables, and Φ is a set of quantifier-free constraints over the variables in X. Conclusions c are predicate applications $n(\vec{z}) \in P_X := \{n(\vec{x}) \mid |\vec{x}| = |\mathcal{S}(n)| \wedge \forall i \in \{1, \ldots, |\vec{x}|\}. \pi_i(\vec{x}) = x \wedge \pi_i(\mathcal{S}(n)) = (D, \leq) \implies (x, D) \in X\}$ over variables in X that respect the variable type. Correspondingly, the premises $P \subseteq P_X$, are a set of predicate applications over variables in X.

We lift the suborders of $\mathcal{S}$ to an order on abstract configurations $\Delta_1, \Delta_2 \in \mathcal{A}_\mathcal{S}$ as follows:

$$\begin{aligned} n_1(\vec{t_1}) \leq_p n_2(\vec{t_2}) &:= n_1 = n_2 \\ &\quad \wedge \ \forall i \in \{1, \ldots, |\vec{t_1}|\}. \pi_i(\vec{t_1}) \leq_{n_1,i} \pi_i(\vec{t_2}) \\ \text{given } \pi_i(\mathcal{S}(n)) &= (D_{n,i}, \leq_{n,i}) \\ \Delta_1 \leq \Delta_2 &:= \forall p_1 \in \Delta_1. \exists p_2 \in \Delta_2. p_1 \leq_p p_2 \end{aligned}$$

Finally, we introduce the notion of soundness for a Horn-clause-based abstraction.

Definition 31. *A Horn-clause-based abstraction* $(\mathcal{D}, \mathcal{S}, \alpha, \Lambda)$ soundly approximates *a small-step semantics* S_s *if*

$$\begin{aligned} \forall (c, c') \in S_s^*.\ \forall \Delta.\ \alpha(c) \leq \Delta \\ \Rightarrow \exists \Delta'.\ \Delta, \Lambda \vdash \Delta' \ \wedge \ \alpha(c') \leq \Delta' \end{aligned} \tag{C.1}$$

This statement requires that, whenever a concrete configuration c' is reachable from configuration c (meaning that (c, c') is contained in the reflexive and transitive closure of S_s, denoted as S_s^*), it shall hold that from all abstractions Δ of c, the Horn clause abstraction allows us to logically derive ($\vdash$) a valid abstraction Δ' of c'. Note that α intuitively yields the most concrete abstraction of a configuration, hence to make the property hold for all possible abstractions of a configuration, we strengthen the property to hold for all abstractions that are more abstract than $\alpha(c)$. The soundness theorem implies that whenever we can show that from some abstraction Δ of a configuration c there is no abstract configuration Δ' derivable such that Δ' abstracts c', then c' is not reachable from c. Consequently, if it is possible to enumerate all abstractions of c', checking non-derivability (as it is supported by the fixed point engines of modern SMT solvers) gives us a procedure for proving unreachability of program configurations.

C.2.2 Relation to Abstract Interpretation

It is possible to phrase the previous characterization in terms of classical abstract interpretation notions. More precisely, we can define a Galois connection (α, γ) between sets of concrete configurations $\mathcal{P}(\mathcal{C})$ (ordered by $\subseteq$) and abstract configurations $\mathcal{A}$ (ordered by $\leq$). To this end, we lift the abstraction function α to sets of configurations in a canonical fashion:

$$\alpha(C) := \bigcup_{c \in C} \alpha(c) \tag{C.2}$$

Next, we define the concretization function based on α:

$$\gamma(\Delta) := \{c \in \mathcal{C} \mid \alpha(c) \leq \Delta\}$$

Lemma 31. *The pair of functions (α, γ) forms a Galois connection between $(\mathcal{P}(\mathcal{C}), \subseteq)$ and $(\mathcal{A}, \leq)$.*

Proof. We need to show for all C and Δ that

$$\alpha(C) \leq \Delta \Leftrightarrow C \subseteq \gamma(\Delta)$$

$\Rightarrow$: Let $\alpha(C) \leq \Delta$. Further let $c \in C$. We show that $c \in \gamma(\Delta)$. By the definition of γ it is sufficient to show that $\alpha(c) \leq \Delta$. Let $p_1 \in \alpha(c)$. We show that there is some $p_2 \in \Delta$ such that $p_1 \leq p_2$. Since $p_1 \in \alpha(c)$ and $c \in C$, we know that $p_1 \in \alpha(C)$ and since $\alpha(C) \leq \Delta$ also that there needs to be some $p_2 \in \Delta$ such that $p_1 \leq p_2$ what concludes the proof.

$\Leftarrow$: Let $C \subseteq \gamma(\Delta)$. Further let $p_1 \in \alpha(C)$. We show that there is some $p_2 \in \Delta$ such that $p_1 \leq p_2$. Since $p1 \in \alpha(C)$ there must be some $c \in C$ such that $p_1 \in \alpha(c)$. And from $C \subseteq \gamma(\Delta)$ we can conclude that $c \in \gamma(\Delta)$ which implies that $\alpha(c) \leq \Delta$. Consequently there needs to be a $p_2 \in \Delta$ such that $p_1 \leq p_2$ what concludes the proof.

□

Now, we can define reachability of concrete configurations and derivability of abstract configurations as the least fixed points of step functions (F_I for concrete configuration steps and F'_{Δ_I} for abstract configuration steps) which describe a collecting semantics (with respect to some initial configuration).

$$F_I(C) := \{c' \mid \exists c \in C.\, (c, c') \in S_s\} \cup I$$

$$F'_{\Delta_I}(\Delta) := \{p \mid \Delta, \Lambda \vdash p\} \cup \Delta_I$$

We obtain the following intuitive correspondences between the different characterizations:

$$(c, c') \in S_s^* \Leftrightarrow c' \in \mathit{lfp}[F_{\{c\}}] \tag{C.3}$$

$$\Delta, \Lambda \vdash \Delta' \Leftrightarrow \Delta' \subseteq \mathit{lfp}[F'_{\Delta}] \tag{C.4}$$

where $\mathit{lfp}[f]$ denotes the least fixed point of a function f.

To ensure that the corresponding least fixed points exists, we need to ensure that the domains $\mathcal{P}(\mathcal{C})$ and $\mathcal{A}$ of the Galois connection form a complete lattice and that both F_I and F'_{Δ_I} are monotone. While $\langle \mathcal{P}(\mathcal{C}), \subseteq, \emptyset, \mathcal{P}(\mathcal{C}), \cup, \cap \rangle$ is the canonical power set lattice, we can easily show $\langle \mathcal{A}, \leq, \emptyset, \Delta, \cup, \cap \rangle$ to also form a complete lattice as $\subseteq$ is a subrelation of $\leq$. While it is trivial to show that F_I is monotone, for F'_{Δ_I} it becomes a proof obligation on Λ:

$$\forall \Delta, \Delta' . \Delta \leq \Delta' \wedge \Delta, \Lambda \vdash p \implies \exists p' . p \leq p' \wedge \Delta', \Lambda \vdash p' \tag{C.5}$$

Using the step functions, we can characterize sound over-approximations as defined in Definition 31 in an alternative fashion. More precisely, we require our approximation to be a *sound upper approximation* [CC04].

Lemma 32. *A Horn-clause-based abstraction* $(\mathcal{D}, \{\leq_{n,i}\}_{(n,i)}, \mathcal{S}, \alpha, \Lambda)$ soundly approximates *a small-step semantics* S_s *iff* Λ *satisfies Equation C.5 and for all* $c \in \mathcal{C}$ *and all* $\Delta \geq \alpha(c)$

$$\alpha(\mathit{lfp}[F_{\{c\}}]) \leq \mathit{lfp}[F'_{\Delta}]$$

Proof. "⇒": Assume Equation C.1 and $f_1 \in \alpha(\mathit{lfp}[F_{\{c\}}])$ for some fact f_1. We show that there exists some fact f_2 such that $f_2 \in \mathit{lfp}[F'_{\Delta}]$ and $f_1 \leq f_2$. By Equation C.2, we know that from $f_1 \in \alpha(\mathit{lfp}[F_{\{c\}}])$ we can conclude that there exists some $c' \in \mathit{lfp}[F_{\{c\}}]$ such that $f_1 \in \alpha(c')$. By Equation C.3, we have that $(c, c') \in S_s^*$ and hence by Equation C.1 we can conclude that there exists some Δ' such that $\Delta, \Lambda \vdash \Delta'$ and $\alpha(c') \leq \Delta'$. With $f_1 \in \alpha(c')$ we get from this that there exists some $f_2 \in \Delta'$ such that $f_1 \leq f_2$. Since $\Delta, \Lambda \vdash \Delta'$, we get from Equation C.4 that $\Delta' \subseteq \mathit{lfp}[F'_{\Delta}]$ and hence also $f_2 \in \mathit{lfp}[F'_{\Delta}]$ which concludes the proof.

"⇐": Assume $\alpha(\mathit{lfp}[F_{\{c\}}]) \leq \mathit{lfp}[F'_{\Delta}]$ and let $(c, c') \in S_s^*$ and $\alpha(c) \leq \Delta$. We show that there is some Δ' such that $\Delta, \Lambda \vdash \Delta'$ and $\alpha(c') \leq \Delta'$. By Equation C.3, we get that $c' \in \mathit{lfp}[F_{\{c\}}]$ and hence also $\alpha(c') \subseteq \alpha(\mathit{lfp}[F_{\{c\}}])$ (by Equation C.2). As $\alpha(\mathit{lfp}[F_{\{c\}}]) \leq \mathit{lfp}[F'_{\Delta}]$ it follows that also $\alpha(c') \leq \mathit{lfp}[F'_{\Delta}]$. Additionally, it follows from Equation C.4 that $\Delta, \Lambda \vdash \mathit{lfp}[F'_{\Delta}]$. This closes our proof. □

Given that F'_{Δ} is monotonic, $\alpha(\mathit{lfp}[F_{\{c\}}]) \leq \mathit{lfp}[F'_{\Delta}]$ can be shown to follow from the one-step characterization below:

$$\alpha \circ F \leq F' \circ \alpha \tag{C.6}$$

(where $F = F_{\emptyset}$ and $F' = F'_{\emptyset}$).

This is because $\alpha \circ F \leq F' \circ \alpha$ implies for all $c \in \mathcal{C}$ and all $\Delta \geq \alpha(c)$ that $\alpha \circ F_{\{c\}} \leq F'_{\Delta} \circ \alpha$ and by the fixed point transfer theorem [CC04] for Galois connections, this result can be lifted to least fixed points. As a consequence for proving Theorem 3, it is sufficient to show that Equation C.5 and Equation C.6 hold.

C.2.3 Analysis Definition (continued)

We overview additional details of the analysis definition in Section 4.2.

The orders on the abstract argument domains of the predicates in Fig. 4.2 are formally defined as follows:

$$\begin{aligned}
\leq_{\hat{D}} &:= \{(\hat{a}, \hat{b}) \mid \hat{b} = \top \vee \hat{a} = \hat{b}\} \\
\leq_{\mathbb{N}} &:= \{(m, n) \mid m = n\} \\
\leq_{\mathbb{B}} &:= \{(a, b) \mid a = b\} \\
\leq_{\mathbb{N} \to \hat{D}} &:= \{(f, g) \mid \forall n \in \mathbb{N}.\ f(n) \leq_{\hat{D}} g(n)\} \\
\leq_{\mathbb{N} \times (\mathbb{N} \to \hat{D})} &:= \{((m, f), (n, g)) \mid m = n \ \wedge\ \forall i < m.\ f(i) \leq_{\hat{D}} g(i)\}
\end{aligned}$$

We assume that the same orders apply to the same argument domains of different predicates.

Some of the partially ordered sets described by the argument domains and their corresponding order, have a supremum, as formally stated in the following lemma:

Lemma 33 (Suprema of argument domains)**.** *The following statements hold:*

- $\forall \hat{a} \in \hat{D}.\ \hat{a} \leq_{\hat{D}} \top$
- $\forall f \in \mathbb{N} \to \hat{D}.\ f \leq_{\mathbb{N} \to \hat{D}} \lambda x.\ \top$

Proof. Both cases are immediate consequences of the definitions. □

Abstract Operations. We formally define abstract operations on values from the abstract argument domains, starting with binary operations on natural numbers: Let $op_{bin} \in \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be a binary operation. We define abstract binary operations as follows:

$$\widehat{\cdot} \in (\mathbb{N} \times \mathbb{N} \to \mathbb{N}) \to \hat{D} \times \hat{D} \to \hat{D}$$

$$\widehat{op_{bin}}\ (\hat{x}, \hat{y}) := \begin{cases} op_{bin}(\hat{x}, \hat{y}) & \hat{x}, \hat{y} \in \mathbb{N} \\ \top & \text{otherwise} \end{cases}$$

Similarly, we can define abstract comparison operators. Let $op_{comp} \in \mathbb{N} \times \mathbb{N} \to \mathbb{B}$ be a comparison operation on natural numbers. We define abstract comparison operations as follows:

$$\widehat{\cdot} \in (\mathbb{N} \times \mathbb{N} \to \mathbb{B}) \to \hat{D} \times \hat{D} \to \mathbb{B}$$

$$\widehat{op_{comp}}\ (\hat{x}, \hat{y}) := \begin{cases} op_{comp}(\hat{x}, \hat{y}) & \hat{x}, \hat{y} \in \mathbb{N} \\ 1 & \text{otherwise} \end{cases}$$

We further define the abstract operations for array access used in Figure 4.3. First, we define the function for extracting a specified fraction of an integer (interpreted as 32-byte word)

$$\cdot_{[\cdot,\cdot]} \in \hat{D} \times \mathbb{N} \times \mathbb{N} \to \hat{D}$$
$$\hat{v}_{[l,r]} := \begin{cases} \left\lfloor \frac{\hat{v}}{256^{31-r}} \right\rfloor \bmod 256^{r-l+1} & l \leq r \ \wedge \ \hat{v} \in \mathbb{N} \\ \top & \text{otherwise} \end{cases}$$

Next, we define the append function:

$$||_{\cdot} \in \hat{D} \times \hat{D} \times \mathbb{N} \to \hat{D}$$
$$\hat{v}||_n\hat{w} := \begin{cases} \hat{w} * 256^n + \hat{v} & \hat{v}, \hat{w} \in \mathbb{N} \\ \top & \text{otherwise} \end{cases}$$

We focused here only on those operations that were used in section 4.2. For a full account of all abstract operations, we refer the reader to our *HoRSt* specification in [ext20].

C.2.4 Proof of Soundness

For proving Theorem 3, we will not make immediate use of the proof strategy presented in subsection C.2.2. Even though we proof monotonicity (Eq. (C.5)) separately, since this facilitates the reasoning in the individual cases, we will, in the end, go for a direct proof of the statement in Theorem 3 proceeding by complete induction of the number of small steps. The reason for that is that for proving our abstraction sound one-step reasoning is not sufficient as we need to argue about execution steps that lie further ahead (hence the use of complete induction).

Auxiliary Notions. In order to prove soundness, we will need to require a stronger form of consistency than the usual annotation consistency (Definition 22) for the execution states of c^* that allows to relate the contract code to the currently executed code.

Definition 32 (Strong Annotation Consistency)**.** *An execution state s is strongly consistent with contract annotation c if it is consistent with c and additionally*

$$\mathit{isRegular}(s) \implies s.\iota.\mathit{code} = c.\mathit{code}$$

In order to formally state soundness, we need to put the minor restriction on the executions that we are considering to be collision-resistant for the contract c^* under analysis. This is as in the case of contract creations, it is theoretically possible (with negligible probability) that (due to a hash collision in the $\mathsf{Kec}(\cdot,\cdot)$ function) a contract with the same address as the contract c^* under analysis is created. In this case, the contract's storage, as well as code, will be overwritten. If such an over-write occurred in the execution of c^* (after giving up the control flow due to the call or by performing a CREATE instruction itself), this would mean that any following execution of c^* would resume in the altered storage, and (even more severely) following a new contract code. Consequently, there is no way of faithfully abstracting the execution of c^* once the control flow

was handed over. In practice, however, the occurrence of such a hash collision can be neglected due to its low probability. Formally, we give a soundness guarantee only for those executions that do not encompass a problematic hash collision.

Lemma 34. *Let* $\Gamma \vDash s_c{::}S \rightarrow^n S' +\!\!+ S$ *be a collision-free execution for* $\mathcal{A}_T$ *such that* $\mathit{address}\,(c^*) \in \mathcal{A}_T$, s *be a regular execution state and* c *be consistent with* s, *and* $s.\sigma(\mathit{address}\,(c^*)).\mathsf{code} = \mathit{code}\,(c^*)$. *Then it holds that*

$$\forall {s'}_{c'} \in S'.\ c'.\mathit{addr} \neq c^*.\mathit{addr} \vee c' = c^*$$

Proof. By induction on the size of S'.

1. Let $|S'| = 1$. Then $S' = [{s'}_{c'}]$. If $\mathit{address}\,(c') \neq \mathit{address}\,(c^*)$ the claim trivially holds. So we assume in the following that $\mathit{address}\,(c') = \mathit{address}\,(c^*)$. In this case we know by Lemma 24 that $c' = c$. So in particular $\mathit{address}\,(c') = \mathit{address}\,(c) = \mathit{address}\,(c^*)$. Since c is consistent with s and also $s.\sigma(\mathit{address}\,(c^*)).\mathsf{code} = \mathit{code}\,(c^*)$ we know that $c = c^*$ and hence also $c^* = c'$.
2. Let $|S'| > 1$. Then $S' = {s'}_{c'}{::}S''$ and $|S''| > 0$. If $\mathit{address}\,(c') \neq \mathit{address}\,(c^*)$ the claim trivially holds. So we assume in the following that $\mathit{address}\,(c') = \mathit{address}\,(c^*)$. We distinguish two cases
   a) If $s' \neq \mathit{EXC}$ then by Definition 20 we know that $\mathsf{codes}_{\{\mathit{address}\,(c^*)\}}(s') = \mathsf{codes}_{\{\mathit{address}\,(c')\}}(s)$ and hence $c' = c^*$ what closes the case.
   b) If $s' = \mathit{EXC}$ then we know by Lemma 19 that one of the following holds:
      i. There exists some s^i such that $\Gamma \vDash s_c{::}S \rightarrow^m {s^i}_{c''}{::}S'' +\!\!+ S$ and $\Gamma \vDash {s^i}_{c''}{::}S'' +\!\!+ S \rightarrow^{n-m} {s'}_{c'}{::}S'' +\!\!+ S$ for $m \leq n$. Then we know that either $\mathit{address}\,(c'') \neq \mathit{address}\,(c^*)$ in which case the claim trivially holds due to Lemma 23, or by Definition 20 that $c'' = c^*$. By Lemma 24 we can then further conclude that $c^* = c'$ what closes the case.
      ii. $S'' = {s^3}_{c_3}{::}S_3$ and $c' = c_3$ (since s' represents the initiation of a transaction that failed at call time). The claim hence follows by the inductive hypothesis.

□

This lemma ensures that during the execution, the address of contract c^* can never be attached to a different code. Given that execution states of contract creation are annotated with (ρ, ϵ) (where ρ is the address of the contract in creation), this definition in particular rules out that the creation code of a contract with address $c^*.\mathit{addr}$ is executed.

While the occurrences of such a colliding contract creation needs to be excluded on executions (since it could be performed by arbitrary contracts), exclusion of DELEGATECALL and CALLCODE extractions only applies to the executions of c^* and can therefore be syntactically enforced on c^*'s contract code. We establish the invariant that we obtain from excluding DELEGATECALL and CALLCODE from c^*'s contract code.

Lemma 35 (Strong Annotation Evolution)**.** *Let* $\Gamma \vDash s_c{::}S \rightarrow^n s'_{c'}{::}S' +\!\!+ S$ *be a collision-free execution for* $\mathcal{A}_T$ *such that* c *is strongly consistent with* s *and address* $(c') \in \mathcal{A}_T$*. Then one of the following holds:*

1. s' *is strongly consistent with* c'
2. $S' = s^1{}_{c'} :: S_1$ *for some* s^1 *and* S_1 *and* $s_1.\iota.\mathsf{code}[s^1.\mu.\mathsf{pc}] \in \{\mathit{DELEGATECALL}, \mathit{CALLCODE}\}$

Proof. Proof by case distinction on S'.

- If $S' = \epsilon$ then we know from annotation persistence (Definition 22) that $c = c'$. Further from execution environment persistence (Lemma 22) we also know that $s.\iota.\mathsf{code} = s'.\iota.\mathsf{code}$. Since the annotation consistency is anyway preserved by the execution (Lemma 21), we are left to show that $s'.\iota.\mathsf{code} = c.\mathit{code}$ what immediately follows from the strong consistency of s with c.
- If $S' = s^1{}_{c_1}{::}S_1$ for some s^1 and S_1 and c_1, then we know by Lemma 17 that s^1 is a call state and that the execution stepped through that call state (Lemma 18). By the definition of call states (Definition 17) we hence know that there is some s^2, c_2 such that $\Gamma \vDash s^1{}_{c_1}{::}S_1 +\!\!+ S \rightarrow s^2{}_{c_2}{::}s^1{}_{c_1}{::}S_1 +\!\!+ S$ and $\Gamma \vDash s^2{}_{c_2}{::}s^1{}_{c_1}{::}S_1 +\!\!+ S \rightarrow^* s'_{c'}{::}s^1{}_{c_1}{::}S_1 +\!\!+ S$. Due to the annotation persistence (Lemma 24) we can conclude that $c' = c_2$. Further we know that $s^1.\iota'.\mathsf{code}[s^1.\mu.\mathsf{pc}] \in \mathit{Inst}_{\mathit{call}}$ (Lemma 15). Hence we can proceed by a case distinction on the different call instructions. For CALL and CREATE it trivially follows from the definition of the small-step rules that s^1 is strongly consistent with c^1. For the cases of DELEGATECALL and CALLCODE it follows from the definition of the small-step rules that $c^1 = c^2$ and hence $c^1 = c'$ what concludes the proof.

□

Lemma 36 (Annotation Evolution)**.** *Let* $(\Gamma, s_c{::}S)$ *be a configuration that is reachable via a collision-free execution for* $\mathcal{A}_T$ *with address* $(c) \in \mathcal{A}_T$*. Then one of the following holds:*

1. s *is strongly consistent with* c
2. $S = s'_c{::}S'$ *for some* s' *and* S' *and* $s'.\iota.\mathsf{code}[s'.\mu.\mathsf{pc}] \in \{\mathit{DELEGATECALL}, \mathit{CALLCODE}\}$

Proof. Since $(\Gamma, s_c{::}S)$ is reachable, we know that there exists some s^i, c^i such that $(\Gamma, s^i{}_{c^i}) = \mathit{initializeT}(T, \sigma, H, g)$ for some transaction T, global state σ, block header H and gas g and $\iota \vDash s^i{}_{c^i}{::}\epsilon \rightarrow^* s_c{::}S$. By definition of *initializeT*, s^i is strongly consistent with c^i. The claim hence immediately follows from Lemma 35. □

Note that this lemma, in particular, states that strong consistency can only be broken by a contract call involving a DELEGATECALL and CALLCODE instruction and that in this case, the contract annotation of the caller executing this instruction persists.

Lemma 37 (Strong Consistency of Reachable States)**.** *Let c be a contract such that $\{\mathit{DELEGATECALL}, \mathit{CALLCODE}\} \cap c.code = \emptyset$. Further, let $\Gamma \vDash s'_{c'} :: S \rightarrow^n s_c :: S' +\!\!+ S$ be a collision-free execution for $\mathcal{A}_T$ such that s' is strongly consistent with c' and address $(c) \in \mathcal{A}_T$. Then s is strongly consistent with c.*

Proof. We proceed by induction on (the size of) the call stack S'.

- If $S' = \epsilon$ then by Lemma 35 we can immediately conclude that s is strongly consistent with c.
- If $S' = s^1{}_{c^1} :: S_1$ for some s^1, c_1, S_1 we know by Lemma 35 that either s^1 is strongly consistent with c^1 or that $c^1 = c$ and $s^1.\iota.\mathsf{code}[s^1.\mu.\mathsf{pc}] \in \{\mathsf{DELEGATECALL}, \mathsf{CALLCODE}\}$. In the first case, the statement trivially holds. For the second case by Lemma 18 we know that $\Gamma \vDash s'_{c'} :: S \rightarrow^m s^1{}_{c_1} :: S_1 +\!\!+ S$ for some m and hence we can apply the inductive hypothesis that gives us that s^1 is strongly consistent with c_1 and hence also with c. By the definition of strong consistency we can conclude that $s^1.\iota.\mathsf{code} = c.code$ and hence that $c.code[s^1.\mu.\mathsf{pc}] \in \{\mathsf{DELEGATECALL}, \mathsf{CALLCODE}\}$ what contradicts the assumption and hence concludes the proof.

□

Lemma 38 (Strong Consistency of Reachable Configurations)**.** *Let c be a contract such that $\{\mathit{DELEGATECALL}, \mathit{CALLCODE}\} \cap c.code = \emptyset$. Further, let $(\Gamma, s_c :: S)$ be a configuration that is reachable via a collision-free execution for $\mathcal{A}_T$ with address $(c) \in \mathcal{A}_T$. Then s is strongly consistent with c.*

Proof. Since $(\Gamma, s_c :: S)$ is reachable, we know that there exists some s^i, c^i such that $(\Gamma, s^i{}_{c^i}) = \mathit{initializeT}(T, \sigma, H, g)$ for some transaction T, global state σ, block header H and gas g and $\iota \vDash s^i{}_{c^i} :: \epsilon \rightarrow^* s_c :: S$. By definition of *initializeT*, s^i is strongly consistent with c^i. The claim hence immediately follows from Lemma 37. □

Lemma 39 (Annotation Agreement for c^*)**.** *Let c be a contract such that $\{\mathit{DELEGATECALL}, \mathit{CALLCODE}\} \cap c.code = \emptyset$. Let further $\Gamma \vDash s_c :: S \rightarrow^n S' +\!\!+ S$ be a collision free execution for $\mathcal{A}_T$ such that address $(c) \in \mathcal{A}_T$ and let s be strongly consistent with c. Then for all (regular) execution states $s'_{c'} \in S'$ it holds that*

1. *If $c.addr = s'.\iota.\mathsf{actor}$ then $c.code = s'.\iota.code$*
2. *If $c' \neq c$ then $s'.\iota.\mathsf{actor} \neq c.addr$*

Proof. By induction on n

1. If $n = 0$ then $S' = [s_c]$. Since we know that s is strongly consistent with c we now that $c.code = s.\iota.\mathsf{code}$. So conditions 1) and 2) hold.

2. If $n > 0$ then (A1) $\Gamma \vDash s_c{::}S \rightarrow^{n-1} S^*$ and (A1) $\Gamma \vDash S^* \rightarrow S' +\!\!+ S$. By Lemma 6 we knot that $S^* = S_3 +\!\!+ S$ for some S_3 with $|S_3| > 0$. By the inductive hypothesis we can conclude that conditions 1) and 2) hold for all ${s'}_{c'} \in S_3$. Since $|S_3| > 0$ we know that $S_3 = {s^3}_{c_3}{::}S_4$ for some ${s^3}_{c_3}$ and S_4. We make a case distinction on the small step execution step performed in (A2)

   - In case of a local execution execution step we know that $S' = {s^4}_{c_3}{::}S_4$ for some s^4. Since local execution steps do not effect the execution environment we know that conditions 1) and 2) hold for ${s^4}_{c_3}$ since (by the inductive hypothesis) they hold for ${s^3}_{c_3}$
   - In case of the initiation of an internal transaction we know that $S' = {s^4}_{c_3}{::}{s^3}_{c_3}{::}S_4$ for some s^4 and c_4. By Lemma 35 we know that we have one of the following cases:

     a) s^4 is strongly consistent with c_4. In this case, we know that (B1) $s^4.\iota.\mathsf{code} = c_4.\mathit{code}$, (B2) $s^4.\iota.\mathsf{actor} = c_4.a$, and (B3) $s^4.\sigma(c_4.a.\mathsf{code} = c_4.\mathit{code}$. We first show condition 1) for ${s^4}_{c_4}$: If $c.a = s^4.\iota.\mathsf{actor}$ then by (B4) also $c.a = c_4.a$. Further by Lemma 34 we have that also $c = c_4$ and consequently $s^4.\iota.\mathsf{code} = c.\mathit{code}$. We next show condition 2) for ${s^4}_{c_4}$: Assume $c_4 \neq c$ then due to Lemma 34 we know that $c_4.a \neq c.a$. By (B2) we can hence conclude that $s^4.\iota.\mathsf{actor} \neq c.a$

     b) $s^3.\iota.\mathsf{code}[s^3.\mu.\mathsf{pc}] \in \{\mathsf{DELEGATECALL}, \mathsf{CALLCODE}\}$. In this case we in particular know that $s^3.\iota.\mathsf{code} \neq c.\mathit{code}$ (since we know c not to contain DELEGATECALL and CALLCODE instruction). By condition (1) of the inductive hypothesis we hence get that $c.a \neq s^3.\iota.\mathsf{actor}$. By Lemma 21 we know that $c_3.a = s^3.\iota.\mathsf{actor}$, so we can also conclude that $c_3.a \neq c.a$. By the semantics of DELEGATECALL and CALLCODE we further know that $s^4.\iota.\mathsf{actor} = s^3.\iota.\mathsf{actor}$. So we can conclude that $s^4.\iota.\mathsf{actor} = s^3.\iota.\mathsf{actor} = c_3.a \neq c.a$.

3. In the case of the completion of an internal transaction we know that $S_4 = {s^4}_{c_4}{::}S_5$ for some s^4, c_4 and S_5 and $S' = {s^5}_{c_5}{::}S_5$ for some s^5 ad c_5. By Lemma 18 we know that $\Gamma \vDash {s^4}_{c_4}{::}S_5 +\!\!+ S \rightarrow^* {s^5}_{c_5}{::}S_5 +\!\!+ S$ and hence by Lemma 22 and Lemma 23 that $s^4.\iota = s^5.\iota$ and $c_4.a = c_5.a$. By the inductive hypothesis we know that conditions 1) and 2) hold for ${s^4}_{c_4}$. We first show condition (1) for ${s^5}_{c_5}$: Assume that $c.a = s^5.\iota.\mathsf{actor}$. Then we know that $c.a = s^4.\iota.\mathsf{actor}$ and hence $c.\mathit{code} = s^4.\iota.\mathit{code} = s^5.\iota.\mathit{code}$ what concludes the case. We next show condition (2) for ${s^5}_{c_5}$: Assume that $c_5 \neq c$. We distinguish two cases:

   a) Assume that $c.a \neq c_5.a$ Since $c_4.a = c_5.a$ we also know that $c.a \neq c_4.a$ and hence (by the inductive hypothesis) that $s^4.\iota.\mathsf{actor} \neq c.\mathit{addr}$. And so also $s^5.\iota.\mathsf{actor} \neq c.\mathit{addr}$

   b) Assume that $c.a = c_5.a$, but $c.\mathit{code} \neq c_5.\mathit{code}$. Since $c_4.a = c_5.a$ we know that then also $c.a = c_4.a$. Hence by Lemma 21 we can conclude that $c_4 = c_5$. By the inductive hypothesis we know in this case however that $s^4.\iota.\mathsf{actor} \neq c.\mathit{addr}$. So we also know that $s^5.\iota.\mathsf{actor} \neq c.\mathit{addr}$.

□

So, in a nutshell, for contracts not containing CALLCODE and DELEGATECALL, strong consistency is preserved, and additionally, the contract code is persistent (cannot change over the execution).

For arguing about the call abstraction, we show the following substantial lemma that allows us to trace back the storage of a (contract) account to the (result of a) prior execution of this contract.

Lemma 40 (Storage Evolution)**.** *Let* Γ, S, S', s, s', *and* $c' \neq c^*$ *be such that* s *is strongly consistent with* c^* *and*

$$\Gamma \vDash s_{c^*} :: S \ \rightarrow^n \ s'_{c'} :: S' +\!\!+ S$$

is a collision-free execution for c^**. Then one of the following holds:*

1. $s' = \mathit{EXC}$
2. $s' \neq \mathit{EXC} \ \wedge$
   $\exists s^*_{c^*} \in S' \ s'.\sigma(c^*.\mathit{addr}).\mathit{stor} = s^*.\sigma(c^*.\mathit{addr}).\mathit{stor}$
3. $s' \neq \mathit{EXC} \ \wedge$
   $\exists \ S^* \ \sigma \ \mathit{gas} \ d \ \eta \ m. \ |S^*| > 0$
   $\wedge \ \Gamma \vDash s_{c^*} :: S \ \rightarrow^m \mathit{HALT}(\sigma, \mathit{gas}, d, \eta) :: S^* +\!\!+ S$
   $\wedge \ \Gamma \vDash \mathit{HALT}(\sigma, \mathit{gas}, d, \eta) :: S^* +\!\!+ S \ \rightarrow^{n-m} s'_{c'} :: S' +\!\!+ S$
   $\wedge \ s'.\sigma(c^*.\mathit{addr}).\mathit{stor} = \sigma(c^*.\mathit{addr}).\mathit{stor}$

This lemma allows for relating the storage of contract c^* to prior executions of c^* itself. More precisely, the storage of c^* (given that c^* does not contain CALLCODE or DELEGATECALL instructions) either needs to be as it was at the point of the last call originating from c^* or the result of some finished prior execution of c^*.

Proof. We proceed by complete induction on the number $n \in \mathbb{N}$ of small-steps.

- Case $n = 0$. In this case it holds that $s'_{c'} = s_{c^*}$ and $S' = \epsilon$. Hence the assumption that $c' \neq c^*$ is trivially violated.
- Case $n > 0$. In this case $\Gamma \vDash s_{c^*} :: S \ \rightarrow^{n-1} S''$ and $\Gamma \vDash S'' \ \rightarrow \ s'_{c'} :: S' +\!\!+ S$ for some S''. We proceed by case analysis on the small-step rule being applied in the last step.
  - ADD. (non exception case). Then $S'' = (\mu, \iota, \sigma, \eta)_{c'} :: S' +\!\!+ S$ for some μ, ι, σ, and η and $s' = (\mu', \iota, \sigma, \eta)$ for some μ'. By inductive hypothesis for $n - 1$ it follows that one out of options 1 to 3 holds for $(\mu, \iota, \sigma, \eta)$. As the global state σ stays unaffected, this consequently also holds for $(\mu', \iota, \sigma, \eta)$ hence closing the case. This reasoning applies to all local rules that are not changing the contracts global storage (so except for SSTORE).

– SSTORE. (non exception case). Then $S'' = (\mu, \iota, \sigma, \eta)_{c'} :: S' +\!\!+ S$ for some μ, ι, σ, and η and $s' = (\mu', \iota, \sigma', \eta)$ for some μ', σ'. Since SSTORE only modifies the storage of the active account, we can conclude that for all addresses a such that $a \neq \iota.\mathsf{actor}$ it holds that $\sigma(a) = \sigma'(a)$. Since by Lemma 39 it holds that $\iota.\mathsf{actor} \neq c^*.addr$, it particularly follows that $\sigma(c^*.addr) = \sigma'(c^*.addr)$. Hence the claim follows immediately from the application of the inductive hypothesis for $n - 1$.

– CALL. (All preconditions satisfied, called account exists). Then $S'' = {s''}_{c''} :: S''' +\!\!+ S$ and $S' = {s''}_{c''} :: S'''$ for some regular execution state s'', contract c'' and call stack S''. We do a case distinction on c'':

$c'' = c^*$ In this case, condition 2 is satisfied since ${s''}_{c''} \in S'$ and the call itself does not affect the contract's storage, so $s'.\sigma(c^*.addr).stor = s''.\sigma(c^*.addr).stor$.

$c'' \neq c^*$ In this case the inductive hypothesis is applicable for $n - 1$. Given again that the call itself does not affect storage, so $s'.\sigma(c^*.addr).stor = s''.\sigma(c^*.addr).stor$, the claim straightforwardly propagates to the case of n steps. Note that similar reasoning also applies to the cases of STATICCALL, CALLCODE, and DELEGATECALL.

– *Halt.* (return from regular halting after CALL) Then $S'' = HALT(\sigma, gas, d, \eta)_{\dot{c}} :: {s''}_{c'} :: S' +\!\!+ S$ for some σ, gas, d, η, s'', and c''. Additionally, it holds that $s'.\sigma = \sigma$. We do a case distinction on $\dot{c}$:

$\dot{c} = c^*$ In this case, condition 3 is satisfied since $\Gamma \vDash s_{c^*} :: S \rightarrow^{n-1} HALT(\sigma, gas, d, \eta)_{\dot{c}} :: {s''}_{c'} :: S' +\!\!+ S$, $\Gamma \vDash HALT(\sigma, gas, d, \eta)_{\dot{c}} :: {s''}_{c'} :: S' +\!\!+ S \rightarrow {s'}_{c'} :: S' +\!\!+ S$ and as $s'.\sigma = \sigma$, also $s'.\sigma(c^*.addr).stor = \sigma(c^*.addr).stor$.

$\dot{c} \neq c^*$ In this case again the inductive hypothesis can be applied for $n - 1$, and since $s'.\sigma = \sigma$, the claim trivially carries over to the case of n steps.

– *Exc.* (return from exceptional halting) Then $S'' = EXC_{\dot{c}} :: {s''}_{c'} :: S' +\!\!+ S$ and $s'.\sigma = s''.\sigma$ (as the global state is rolled back). By Lemma 18, we know that there exists some $m < n - 1$ such that $\Gamma \vDash s_{c^*} :: S \rightarrow^{m} {s''}_{c'} :: S' +\!\!+ S$ and $\Gamma \vDash {s''}_{c'} :: S' +\!\!+ S \rightarrow^{n-1-m} EXC_{\dot{c}} :: {s''}_{c'} :: S' +\!\!+ S$. By applying the inductive hypothesis for m ($< n$), the claim straightforwardly carries over to the case of n steps.

– CREATE. (All preconditions satisfied, created account does not exist (no hash collision). Then $S'' = {s''}_{c''} :: S''' +\!\!+ S$ and $S' = {s''}_{c''} :: S'''$ for some regular execution state s'', contract c'' and callstack S''. The same reasoning as for the CALL case applies.

– CREATE. (All preconditions satisfied, created account exists (hash collision). Then $S'' = {s''}_{c''} :: S''' +\!\!+ S$ and $S' = {s''}_{c''} :: S'''$ for some regular execution state s'', contract c'' and call stack S''. Additionally, we know that $c' = (\rho, \bot)$ where ρ is the newly created address. Here, we need to make use of the assumption that the newly created address ρ is not colliding with the address of c^* ($\rho \neq c^*.addr$). This is ensured as otherwise the execution would not be collision-free (for $\rho = c^*.addr$ the condition of Lemma 34 would be violated for ${s'}_{c'}$). We do a case distinction on c'':

$c'' = c^*$ In this case, condition 2 is satisfied since ${s''}_{c''} \in S'$ and the contract creation does not affect the storage of address $c^*.addr$ (but only the one of ρ, which is different by assumption). So $s'.\sigma(c^*.addr).stor = s''.\sigma(c^*.addr).stor$.

$c'' \neq c^*$ In this case the inductive hypothesis is applicable for $n-1$. Given again that the contract creation does not affect the storage of address $c^*.addr$, so $s'.\sigma(c^*.addr).stor = s''.\sigma(c^*.addr).stor$, the claim straightforwardly propagates to the case of n steps.

– *Halt.* (return from regular halting after CREATE) Similar to the halting case for CALL with the only difference that instead of $s'.\sigma = \sigma$ it only holds that $s'.\sigma = \sigma\langle\rho \to acc\rangle$ for some account state acc. However, as long as it is ensured that $\rho \neq c^*.addr$ (which is the case due to the collision-free execution) this does not affect the reasoning.

□

Monotonicity of Abstract Rules. We prove separately, that all rules in $\delta(c^*)$ are monotone (for any c^*). This facilitates the reasoning in the individual cases of the main proof since it allows us to argue about most concrete abstractions only.

Since monotonitiy is independent of the small-step semantics, we will in the following consider an abstract semantics specified by $(\mathcal{D}, \mathcal{S}, \Lambda)$. First, we define monotonicity for an abstract semantics $(\mathcal{D}, \mathcal{S}, \Lambda)$ as follows:

Definition 33 (Monotonicity of Abstract Semantics)**.** *An abstract semantics $(\mathcal{D}, \mathcal{S}, \Lambda)$ is monotone if for all abstract configurations Δ_I, Δ_I', $\Delta_F \in \mathcal{A}_\mathcal{S}$ such that $\Delta_I \leq \Delta_I'$ it holds that*

$$\Delta_I \cup \Lambda \vdash \Delta_F \implies \exists \Delta_F'.\ \Delta_I' \cup \Lambda \vdash \Delta_F' \wedge \Delta_F \leq \Delta_F'$$

We will prove the following theorem:

Theorem 7 (Monotonicity of δ)**.** *For all contracts c it holds that $(\mathcal{D}_{evm}, \mathcal{S}_{evm}, \delta(c))$ is monotone. (Where $\mathcal{D}_{evm}$ is the super domain and $\mathcal{S}_{evm}$ is the signature induced by the definition in Figure 4.2.)*

We prove this property by proving the (one-step) monotonicity of the individual rules in $\delta(c)$.

We define one-step derivations of a Horn clause H from some abstract configuration Δ. To this end, we use the notion of a variable assignment $V \in \mathit{Vars} \to \mathcal{D}$ that maps variables to values of the corresponding abstract domain. We write $V(n(\vec{z}))$ for $n(V(\vec{z}))$ and $V(\{f_1, \ldots, f_n\})$ for $\{V(f_1), \ldots, V(f_n)\}$. By $V \vDash \Phi$ we denote that replacing all variables in Φ according to V yields a tautology.

Definition 34 (One-step Derivability from Horn Clause)**.** *Let $(\mathcal{D}, \mathcal{S}, \Lambda)$ be an abstract semantics and $(\forall X.\ \Phi, P \Rightarrow c) \in \Lambda$. Further let $f \in \mathcal{A}_\mathcal{S}$ Then the one-step derivability relation $\vdash^1$ on abstract configurations is defined as follows:*

$$\Delta, (\forall X.\ \Phi, P \Rightarrow c) \vdash^1 f := \exists V.\ V(P) \subseteq \Delta \ \wedge\ V \vDash \Phi \ \wedge\ f = V(c)$$

Note that this intuition implicitly enforces that the valuation V respects the argument types of the predicates.

We extend the notion of derivability to sets of horn clauses and abstract configurations:

Definition 35 (One-step Derivability from Abstract Semantics)**.** *Let $(\mathcal{D}, \mathcal{S}, \Lambda)$ be an abstract semantics. Then the one-step derivability relation $\vdash^1$ on Λ is defined as follows*

$$\Delta, \Lambda \vdash^1 \Delta' := \exists f.\ \Delta' = \Delta \cup \{f\} \ \wedge\ \exists H \in \Lambda.\ \Delta, H \vdash^1 f$$

Finally, we define $\vdash$ to be the reflexive, transitive closure of $\vdash^1$.

We define the monotonicity of a Horn clause as follows:

Definition 36 (Monotonicity of Horn Clauses)**.** *Let $(\mathcal{D}, \mathcal{S}, \Lambda)$ be an abstract semantics. A constrained Horn clause $H \in \Lambda$ is monotone if for all $\Delta' \geq \Delta$*

$$\Delta, H \vdash^1 f \implies \exists f'.\ \Delta', H \vdash^1 f' \ \wedge f' \geq f$$

Evidently, the (one-step) monotonicity of all Horn clauses in an abstract semantics implies the (multi-step) monotonicity of the abstract semantics

Lemma 41. *Let $(\mathcal{D}, \mathcal{S}, \Lambda)$ be an abstract semantics. If all constrained horn clauses $H \in \Lambda$ are monotone, then so is Λ.*

Proof. Assume that all $H \in \Lambda$ are monotone. Let $\Delta'_I \geq \Delta_I$ and $\Delta_I \cup \Lambda \vdash \Delta_F$. We show that $\exists \Delta'_F.\ \Delta'_I \cup \Lambda \vdash \Delta'_F \wedge \Delta_F \leq \Delta'_F$. From $\Delta_I \cup \Lambda \vdash \Delta_F$ we know that there exists an n step derivation $\Delta_I \cup \Lambda \vdash^n \Delta_F$. We proceed by induction on n

1. If $n = 0$ then $\Delta_F = \Delta_I$. Hence the claim trivially holds since $\Delta'_I \cup \Lambda \vdash \Delta'_I$ and $\Delta_I \leq \Delta'_I$.
2. If $n > 0$ then $\Delta_I \cup \Lambda \vdash^{n-1} \Delta$ and $\Delta \cup \Lambda \vdash^1 \Delta_F$. By the inductive hypothesis we know that $\exists \Delta'.\ \Delta'_I \cup \Lambda \vdash \Delta' \wedge \Delta \leq \Delta'$. Further we know that there is some f such that $\Delta_F = \Delta \cup \{f\}$ and there is some $H \in \Lambda$ such that $\Delta, H \vdash^1 f$. Since we know that H is monotone we know that there is some f' such that $\Delta', H \vdash^1 f'$ and $f' \geq f$. Consequently we know that for $\Delta'_F = \Delta' \cup \{f'\}$ we have that $\Delta' \cup \Lambda \vdash \Delta'_F$ and hence also $\Delta_I \cup \Lambda \vdash \Delta'_F$. Further we know that $\Delta'_F \geq \Delta_F$ since $\Delta' \geq \Delta$ and $f' \geq f$.

□

It is hence sufficient to prove the (one-step) monotonicity of all Horn clauses in $(\mathcal{D}_{evm}, \mathcal{S}_{evm}, \delta(c))$ (for arbitrary c).

For facilitating the proofs, we give a more syntactic characterization of Horn clause monotonicity:

Lemma 42. *Let $H = \forall X.\ \Phi, P \Rightarrow c$ be a Horn clause. If for all variable assignments V, V' with $(x, D) \in X \implies V(x) \in D \ \wedge V'(x) \in D$ it holds that*

$$V'(P) \geq V(P) \wedge V \vDash \Phi$$
$$\implies \exists V^*.\ V^*(P) = V'(P) \wedge V^*(c) \geq V(c) \wedge V^* \vDash \Phi$$

then H is monotone.

Proof. Assume that (1)

$$V'(P) \geq V(P) \wedge V \vDash \Phi$$
$$\implies \exists V^*.\, V^*(P) = V'(P) \wedge V^*(c) \geq V(c) \wedge V^* \vDash \Phi$$

holds for valuations as defined above. We show the monotonicity of $H = \forall X.\, \Phi, P \Rightarrow c$. To this end we assume some (2) $\Delta \geq \Delta$ and (3) $\Delta, H \vdash^1 f$ and show that there is some valuation V' such that $V'(P) \subseteq \Delta'$, $V' \vDash \Phi$ and $V'(c) \geq f$. From (3) it is known that there is some valuation V such that $V(P) \subseteq \Delta$, $V \vDash \Phi$ and $f = V(c)$. From (2), we get that for every $p \in V(P)$ there exists a $p' \in \Delta'$ such that $p \leq p'$. Given that the variables of all premises are distinct, we can easily construct a valuation V' such that $V'(q) = p$ for some $q \in P$ and consequently $V'(P) \subseteq \Delta'$ and $V(P) \leq V'(P)$. Using (1), we get that there is some V^* such that $V^*(P) = V'(P)$ and $V^*(c) \geq V(c)$ and $V^* \vDash \Phi$. Consequently, since $V^*(P) = V'(P) \subseteq \Delta'$ and $V^*(c) \geq V(c) = f$, V^* satisfies all required conditions. □

This lemma reduces proving monotonicity of the constrained Horn clause to proving the monotonicity of the clause's constraints.

Abstract Operations. We exemplary show the monotonicity of the rules shown in Figure 4.4. To this end, we will first establish some general monotonicity results on abstract operations.

Lemma 43 (Monotonicity of Abstract Binary Operations)**.** *Let $\hat{x}$, $\hat{x'}$, $\hat{y}$, $\hat{y'}$ such that $\hat{x} \leq \hat{x'}$ and $\hat{y} \leq \hat{y'}$. Then*

$$\widehat{op_{bin}}\ (\hat{x}, \hat{y}) \leq_{\hat{D}} \widehat{op_{bin}}\ (\hat{x'}, \hat{y'})$$

Proof. We perform a case distinction on $\hat{x} \leq \hat{x'}$.

- If $\hat{x'} = \top$ then $\widehat{op_{bin}}\ (\hat{x'}, \hat{y'}) = \top$ and consequently $\widehat{op_{bin}}\ (\hat{x}, \hat{y}) \leq_{\hat{D}} \widehat{op_{bin}}\ (\hat{x'}, \hat{y'})$
- If $\hat{x} = x \in \mathbb{N}$ and $\hat{x'} = x$ then we do a case distinction on $\hat{y} \leq \hat{y'}$
  - If $\hat{y'} = \top$ then $\widehat{op_{bin}}\ (\hat{x'}, \hat{y'}) = \top$ and consequently $\widehat{op_{bin}}\ (\hat{x}, \hat{y}) \leq_{\hat{D}} \widehat{op_{bin}}\ (\hat{x'}, \hat{y'})$.
  - If $\hat{y} = y \in \mathbb{N}$ and $\hat{y'} = y$ then we know that $\widehat{op_{bin}}\ (\hat{x}, \hat{y}) = op_{bin}(x, y) = \widehat{op_{bin}}\ (\hat{x'}, \hat{y'})$ and hence $\widehat{op_{bin}}\ (\hat{x}, \hat{y}) \leq_{\hat{D}} \widehat{op_{bin}}\ (\hat{x'}, \hat{y'})$.

□

Lemma 44 (Monotonicity of Abstract Comparison Operations)**.** *Let $\hat{x}$, $\hat{x'}$, $\hat{y}$, $\hat{y'}$ such that $\hat{x} \leq_{\hat{D}} \hat{x'}$ and $\hat{y} \leq_{\hat{D}} \hat{y'}$. Then*

$$\widehat{op_{comp}}\ (\hat{x}, \hat{y}) = 1 \implies \widehat{op_{comp}}\ (\hat{x'}, \hat{y'}) = 1$$

Proof. We perform a case distinction on $\hat{x} \leq \hat{x'}$.

- If $\hat{x}' = \top$ then $\widehat{op_{comp}}\ (\hat{x}', \hat{y}') = 1$ and consequently the implication trivially holds.
- If $\hat{x} = x \in \mathbb{N}$ and $\hat{x}' = x$ then we do a case distinction on $\hat{y} \leq \hat{y}'$
  - If $\hat{y}' = \top$ then $\widehat{op_{comp}}\ (\hat{x}', \hat{y}') = 1$ and consequently the implication trivially holds.
  - If $\hat{y} = y \in \mathbb{N}$ and $\hat{y}' = y$ then we know that $\widehat{op_{comp}}\ (\hat{x}', \hat{y}') = op_{comp}(x, y) = \widehat{op_{comp}}\ (\hat{x}, \hat{y})$ and hence the implication holds.

□

Lemma 45 (Monotonicity of Memory Access)**.** *Let* $m_1, m_2 \in \mathbb{N} \to \hat{D}$ *such that* $m_1 \leq_{\mathbb{N}\to\hat{D}} m_2$ *and let* $p \in \mathbb{N}$.

$$getWord(m_1, p) \leq_{\hat{D}} getWord(m_2, p)$$

Proof. The claim immediately follows from the definition of $\leq_{\mathbb{N}\to\hat{D}}$. □

We now illustrate how to prove Theorem 7, using Lemma 42.

Proof. For showing the monotonicity of $\delta(c^*)$ for arbitrary c^* it is sufficient to show the one-step derivability of all rules in $(\!|inst|\!)_{pc}$ for all instructions *inst* and an arbitrary program counter *pc*. Hence, let $pc \in \mathbb{N}$ be arbitrary. The proof proceeds by case distinction on the instruction set.

- ADD. We now prove the monotonicity of the rules for addition in Figure 4.4. Recall the definition of the clause for addition.

$$\begin{aligned}&\mathsf{MState}_{\mathsf{pc}}((size, s), m, stor, c\ell)\ \wedge\ size > 1\\ &\wedge\ \hat{x} = s[size - 1]\ \wedge\ \hat{y} = s[size - 2]\\ &\Longrightarrow\ \mathsf{MState}_{\mathsf{pc}+1}((size - 1, s[size - 2 \to \hat{x} \mathbin{\widehat{+}} \hat{y}]), m, stor, c\ell)\end{aligned}$$

  We prove the monotonicity using Lemma 43. Assume that there is some variable assignment satisfying the rule constraints, meaning that there are values $(size, s)$, m, $stor$, $c\ell$, $\hat{x}$, $\hat{y}$ satisfying $size > 1$, $\hat{x} = s[size - 1]$ and $\hat{y} = s[size - 2]$. We show for any values $(size', \mathsf{s}') \geq_{\mathbb{N}\times(\mathbb{N}\to\hat{D})} (size, s)$, $m' \geq_{\mathbb{N}\to\hat{D}} m$, $stor' \geq_{\mathbb{N}\to\hat{D}} stor$, $c\ell' \geq_{\mathbb{B}} c\ell$ that there are $\hat{x}'$, $\hat{y}'$ such that $size' > 1$, $\hat{x}' = s'[size' - 1]$ and $\hat{y}' = s'[size' - 2]$, and $(size - 1, s[size - 2 \to \hat{x} \mathbin{\widehat{+}} \hat{y}]) \leq_{\mathbb{N}\times(\mathbb{N}\to\hat{D})} (size' - 1, s'[size' - 2 \to \hat{x}' \mathbin{\widehat{+}} \hat{y}'])$. First we observe that $size = size'$ and (since $(size, s) \leq_{\mathbb{N}\times(\mathbb{N}\to\hat{D})} (size', s')$). We pick $\hat{x}' = s'[size - 1]$ and $\hat{y}' = s'[size - 2]$ and from $(size, s) \leq_{\mathbb{N}\times(\mathbb{N}\to\hat{D})} (size', s')$ we know that $s[size - 1] \leq_{\hat{D}} s'[size - 1]$ and $s[size - 2] \leq_{\hat{D}} s'[size - 2]$, so consequently also $\hat{x} \leq_{\hat{D}} \hat{x}'$ and $\hat{y} \leq_{\hat{D}} \hat{y}'$. So we are left to show that $(size - 1, s[size - 2 \to \hat{x} \mathbin{\widehat{+}} \hat{y}]) \leq_{\mathbb{N}\times(\mathbb{N}\to\hat{D})} (size' - 1, s'[size - 2 \to \hat{x}' \mathbin{\widehat{+}} \hat{y}'])$. Since $(size, s) \leq_{\mathbb{N}\times(\mathbb{N}\to\hat{D})} (size', s')$, we only need to show that $\hat{x} \mathbin{\widehat{+}} \hat{y} \leq_{\hat{D}} \hat{x}' \mathbin{\widehat{+}} \hat{y}'$ which immediately follows from Lemma 43. The same reasoning applies to all other binary operations or comparison operations.

- MLOAD. Recall the definition of the rule for memory access:

$$\begin{aligned}&\mathsf{MState}_{\mathsf{pc}}((\mathit{size}, s), m, \mathit{stor}, c\ell) \ \wedge \ \mathit{size} > 1\\ &\wedge \ \hat{o} = s[\mathit{size}-1] \ \wedge \hat{v} = (\hat{o} \in \mathbb{N})\,?\,\mathit{getWord}(m, \hat{o}) \ : \ \top\\ &\implies \mathsf{MState}_{\mathsf{pc}+1}((\mathit{size}, s[\mathit{size}-1 \to \hat{v}]), m, \mathit{stor}, c\ell)\end{aligned}$$

  We prove the monotonicity using Lemma 45. Assume that there is some variable assignment satisfying the rule constraints, meaning that there are values (size, s), m, stor, $c\ell$, $\hat{o}$, $\hat{v}$ satisfying $\mathit{size} > 0$, $\hat{o} = s[\mathit{size}-1]$, and $\hat{v} = (\hat{o} \in \mathbb{N})\,?\,\mathit{getWord}(m, \hat{o}) \ : \ \top$. We show for any values $(\mathit{size}', \mathsf{s}') \geq_{\mathbb{N}\times(\mathbb{N}\to\hat{D})} (\mathit{size}, s)$, $m' \geq_{\mathbb{N}\to\hat{D}} m$, $\mathit{stor}' \geq_{\mathbb{N}\to\hat{D}} \mathit{stor}$, $c\ell' \geq_{\mathbb{B}} c\ell$ that there are $\hat{o}'$, $\hat{v}'$ such that $\mathit{size}' > 1$, $\hat{o}' = s'[\mathit{size}'-1]$ and $\hat{v}' = (\hat{o}' \in \mathbb{N})\,?\,\mathit{getWord}(m, \hat{o}') \ : \ \top$, and $s[\mathit{size}-1 \to \hat{v}] \leq_{\mathbb{N}\times(\mathbb{N}\to\hat{D})} s'[\mathit{size}'-1 \to \hat{v}']$. First we observe $\mathit{size} = \mathit{size}'$ and $c\ell = c\ell'$. We pick $\hat{o}' = s'[\mathit{size}'-1]$ and $\hat{v}' = (\hat{o}' \in \mathbb{N})\,?\,\mathit{getWord}(m', \hat{o}') \ : \ \top$. We know that $s[\mathit{size}-1] \leq_{\hat{D}} s'[\mathit{size}'-1]$ since $(\mathit{size}', \mathsf{s}') \geq_{\mathbb{N}\times(\mathbb{N}\to\hat{D})} (\mathit{size}, s)$ and hence also $\hat{o} \leq_{\hat{D}} \hat{o}'$. For showing that $(\mathit{size}, s[\mathit{size}-1 \to \hat{v}]) \leq_{\mathbb{N}\times(\mathbb{N}\to\hat{D})} (\mathit{size}, s'[\mathit{size}'-1 \to \hat{v}'])$ it is sufficient to show that $\hat{v} \leq_{\hat{D}} \hat{v}'$. We make a case distinction on $\hat{o} \in \mathbb{N}$

  $\hat{o} \in \mathbb{N}$ In this case $\hat{v} = \mathit{getWord}(m, \hat{o})$. Since $\hat{o} \leq_{\hat{D}} \hat{o}'$ we know that either $\hat{o}' = \hat{o}$ or $\hat{o}' = \top$.

  $\hat{o}' = \hat{o}$ In this case clearly $\hat{o}' \in \mathbb{N}$ and hence $\hat{v}' = \mathit{getWord}(m', \hat{o}')$. Since $m \leq_{\mathbb{N}\to\hat{D}} m'$, we know from Lemma 45 that $\mathit{getWord}(m, \hat{o}) \leq_{\hat{D}} \mathit{getWord}(m', \hat{o}')$ and hence $\hat{v} \leq_{\hat{D}} \hat{v}'$.

  $\hat{o}' = \top$ In this case $\hat{v}' = \top$. Since $\top$ is the top element of $\hat{D}$ (Lemma 33), trivially $\hat{v} \leq_{\hat{D}} \hat{v}'$.

  $\hat{o} = \top$ In this case $\hat{v} = \top$ and since $\hat{o} \leq_{\hat{D}} \hat{o}'$ also $\hat{o}' = \top$ and hence $\hat{v}' = \top$ and consequently $\hat{v} \leq_{\hat{D}} \hat{v}'$.

The proof for all other cases is fully analogous. Note in particular that for those rules that are independent of abstract values, monotonicity trivially holds. For example, in the rules for transaction initiating instructions, the conclusions are independent of the (abstract) arguments in the premise, and hence those are trivially monotone. □

Soundness of Abstract Operations. In addition to their monotonicity, we are also interested in the soundness of abstract operations. Intuitively, an abstract operation is sound, if its result is at least as abstract as the result of the concrete operation We formally state soundness for binary operations and comparison operations.

Lemma 46 (Soundness of Abstract Binary Operations)**.** *Let* $x, y \in \mathbb{N}$. *Then*

$$\mathit{op}_{\mathit{bin}}(x, y) \leq_{\hat{D}} \widehat{\mathit{op}_{\mathit{bin}}}\,(x, y)$$

Proof. The claim trivially follows since $\widehat{\mathit{op}_{\mathit{bin}}}\,(x, y) = \mathit{op}_{\mathit{bin}}(x, y)$ for $x, y \in NN$. □

Lemma 47 (Soundness of Abstract Comparison Operations)**.** *Let* $x, y \in \mathbb{N}$. *Then*

$$\mathit{op}_{\mathit{comp}}(x, y) = 1 \implies \widehat{\mathit{op}_{\mathit{comp}}}\,(x, y) = 1$$

Proof. The claim trivially follows since $\widehat{op_{comp}}\,(x,y) = op_{comp}(x,y)$ for $x, y \in NN$. □

That the memory access is sound, is captured by the following lemma:

Lemma 48 (Soundness of Memory Access)**.** *Let* $m \in \mathbb{N} \to \mathbb{N}$ *and* $p \in \mathbb{N}$.

$$m[p]||_1 m[p+1]||_1 \cdots ||_1 m[p+31] \leq_{\hat{D}} \mathit{getWord}(\mathsf{toWordMem}(m), p)$$

Proof. Let $m \in \mathbb{N} \to \mathbb{N}$ and $p \in \mathbb{N}$. We distinguish two cases

1. If $p \bmod 32 = 0$ then we know that $p = 32 \cdot x$ for some $x \in \mathbb{N}$. Consequently we know that $\mathit{getWord}(\mathsf{toWordMem}(m), p) = \mathsf{toWordMem}(m)[x]$. By the definition of $\mathsf{toWordMem}$ we know that this is $m[x \cdot 32]||_1 m[x \cdot 32 + 1] \cdots ||_1 m[x \cdot 32 + 31]$ and hence $\mathit{getWord}(\mathsf{toWordMem}(m), p) = m(p)||_1 m(p+1)||_1 \cdots ||_1 m(p+31)$ What shows the claim.
2. If $p \bmod 32 \neq 0$ then we know that $p = 32 \cdot x + k$ for some $x, k \in \mathbb{N}$ and $k > 0$. Then we can show:

   $$\begin{aligned}
   &\mathit{getWord}(\mathsf{toWordMem}(m), p)\\
   &= (\mathsf{toWordMem}(m)[x]_{[k,31]})||_k(\mathsf{toWordMem}(m)[x+1]_{[0,k-1]})\\
   &= (m[x \cdot 32]||_1 m[x \cdot 32 + 1] \cdots ||_1 m[x \cdot 32 + 31]_{[k,31]})\\
   &\quad ||_k(m[(x+1) \cdot 32]||_1 m[(x+1) \cdot 32 + 1] \cdots ||_1 m[(x+1) \cdot 32 + 31]_{[0,k-1]})\\
   &= (m[x \cdot 32 + k]||_1 \cdots ||_1 m[x \cdot 32 + 31])||_k(m[(x+1) \cdot 32]||_1 \cdots ||_1 m[(x+1) \cdot 32 + k - 1])\\
   &= (m[p]||_1 \cdots ||_1 m[p + 31 - k])||_k(m[p + 32 - k]||_1 \cdots ||_1 m[p + 31])\\
   &= m[p]||_1 \cdots ||_1 m[p + 31]
   \end{aligned}$$

   This concludes the proof.

□

Main Proof. We slightly refine Theorem 3 to consider collision-free executions of c^*, a detail that we omitted in the original formulation for the sake of presentation.

Theorem (Soundness)**.** *Let* c^* *be a contract whose code does not contain* DELEGATECALL *or* CALLCODE*. Let* Γ *be a transaction environment and let* S *and* S' *be annotated call stacks such that* $|S'| > 0$*. Then for all execution states* s *that are strongly consistent with* c^* *such that* $\Gamma \vDash s_{c^*} :: S \to^* S' +\!\!+ S$ *is a collision-free execution, it holds that*

$$\forall \Delta_I.\ \alpha_{c^*}([s_{c^*}]) \leq \Delta_I \implies \exists \Delta.\ \Delta_I, \delta(c^*) \vdash \Delta \ \wedge\ \alpha_{c^*}(S') \leq \Delta$$

We will detail out the relevant cases of the soundness proof. The small-step semantics describes rules for local executions, the initiation of internal transactions, and the completion of internal transactions. For this reason, it suffices to reason about these categories of small-step rules. We do so using the concrete examples of the ADD instruction and the CALL instruction. However, the same reasoning follows for all other instructions of the EVM instruction set.

Proof. By complete induction on the number n of small-steps.

- Case $n = 0$. In the case of the empty reduction sequence, we have that $S' = [s_{c^*}]$ and consequently the claim trivially follows by the reflexivity of $\vdash$.

- Case $n > 0$. Let $\Gamma \vDash s_{c^*} :: S \rightarrow^{n-1} S''$ and $\Gamma \vDash S'' \rightarrow S' +\!\!+ S$. By Lemma 13, it holds that $S'' = S^* +\!\!+ S$ for some S^* with $|S^*| > 0$. By the inductive hypothesis we know that for all $\Delta_I \geq \alpha_{c^*}([s_{c^*}])$ there is some $\Delta_{S^*} \geq \alpha_{c^*}(S^*)$ such that $\Delta_I \cup \delta(c^*) \vdash \Delta_{S^*}$. Consequently, for proving the claim, it is sufficient to show that there is some $\Delta_{S'} \geq \alpha_{c^*}(S')$ such that $\Delta_{S^*} \cup \delta(c^*) \vdash \Delta_{S'}$. As $|S^*| > 0$, we know that $S^* = s'_{c'} :: S^{**}$ for some execution state s', contract c' and call stack S^{**}. The proof is by case analysis on the rule applied in the last reduction step. We show here exemplary the cases for arithmetic operations as well as the rule for calling.

  - ADD (non exception case). Then $s' = (\mu, \iota, \sigma, \eta)$, $\iota.\mathsf{code}\,[\mu.\mathsf{pc}] = \mathsf{ADD}$ and $S' = (\mu', \iota, \sigma, \eta)_{c'} :: S^{**}$. We distinguish the two cases on whether the top stack element $s'_{c'}$ is translated or not ($c' = c^*$)

$c' \neq c^*$ In this case $\alpha_{c^*}(S^*) = \alpha_{c^*}(S^{**})$. As ADD is a local instruction, we know that $S' = s''_{c'} :: S^{**}$ and hence also $\alpha_{c^*}(S') = \alpha_{c^*}(S^{**})$. The claim hence follows trivially from the reflexivity of $\vdash$. The same reasoning applies to all other local instructions.

$c' = c^*$ In this case $\alpha_{c^*}(S^*) = \alpha_s(s', c^*.addr, c\ell) \cup \alpha_{c^*}(S^{**})$ for some $c\ell \in \mathbb{B}$. As s' is strongly consistent with c^* (by Lemma 39), we know that $\iota.code = c^*.code$ and hence $\delta(c^*) \supseteq (\!|\mathsf{ADD}|\!)_{\mu.\mathsf{pc}}$. The claim then follows from the monotonicity of $\delta(c^*)$ (Theorem 7) and the soundness of abstract addition (Lemma 46). The same argumentation applies to all other local operations.

  - CALL (all preconditions satisfied, called account exists). Then $s' = (\mu, \iota, \sigma, \eta)$, $\iota.\mathsf{code}\,[\mu.\mathsf{pc}] = \mathsf{CALL}$ and $S' = (\mu', \iota', \sigma', \eta)_{\dot{c}} :: S^*$ such that μ' is initial, and $\sigma(a).stor = \sigma'(a).stor$ for all addresses a. Again we distinguish the cases whether the newly pushed call stack element $(\mu', \iota', \sigma', \eta)_{\dot{c}}$ is abstracted by α or not.

$\dot{c} \neq c^*$ Then $\alpha_{c^*}(S') = \alpha_{c^*}(S^*)$ and the claim trivially holds.

$\dot{c} = c^*$ We do another case distinction on whether $c' = c^*$

$c' = c^*$ In this case, we know that $\alpha_s(s', addr.c^*, c\ell) \leq \Delta_{S^*}$ (where $c\ell = (S^{**} \neq \epsilon)$). Since s' is strongly consistent with c^* (by Lemma 39), we have that $\iota.code = c^*.code$ and hence $(\!|\mathsf{CALL}|\!)_{\mu.\mathsf{pc}} \subseteq \delta(c^*)$. Since s' is a call state, we have that $(\!|\mathsf{CALL}|\!)_{\mu.\mathsf{pc}} \cup \alpha_s(s', addr.c^*, c\ell) \vdash \{\mathsf{MState}_0((0, \lambda x.\,0), \lambda x.\,0, \sigma(c^*.addr).stor, 1)\}$. As μ' is initial and $\sigma(a).stor = \sigma'(a).stor$, we know additionally that $\{\mathsf{MState}_0((0, \lambda x.\,0), \lambda x.\,0, \sigma(c^*.addr).stor, 1)\} = \alpha_s((\mu', \iota', \sigma', \eta), c^*.addr, c\ell')$ (for $c\ell' = (S^* \neq \epsilon)$). By the monotonicity of $\delta(c^*)$ (Theorem 7), we know that there is also some $\Delta_x \geq \alpha_s((\mu', \iota', \sigma', \eta), c^*.addr, c\ell')$ such that $\Delta_{S^*}, \delta(c^*) \vdash \Delta_x$

which concludes the proof since

$$\begin{aligned}\Delta_{S^*} \cup \delta(c^*) \vdash \Delta_x \cup \Delta_{S^*} \\ \geq \alpha_s((\mu', \iota', \sigma', \eta), c^*.addr, c\ell') \cup \alpha(S^*) \\ = \alpha(S')\end{aligned}$$

$c' \neq c^*$ By Lemma 40, we know (since s' is a regular execution state) that either (1) there exists some $s^*_{c^*} \in S^{**}$ such that $s'.\sigma(c^*.addr).stor = s^*.\sigma(c^*.addr).stor$ or (2) there exist $S^\dagger$, σ^*, gas^*, d^*, η^*, and $m < n$ such that $\Gamma \vDash s_{c^*}::S \rightarrow^m HALT(\sigma^*, gas^*, d^*, \eta^*)_{c^*}::S^\dagger +\!\!+ S$ and $\Gamma \vDash HALT(\sigma^*, gas^*, d^*, \eta^*)_{c^*}::S^\dagger +\!\!+ S \rightarrow^{n-1-m} (\mu', \iota', \sigma', \eta)_{\dot{c}}::s'_{c'}::S^{**} +\!\!+ S$ and $s'.\sigma(c^*.addr).stor = \sigma^*(c^*.addr).stor$. Additionally, we know that then $\sigma^*(c^*.addr).stor = \sigma'(c^*.addr).stor$. We make a distinction on the previously mentioned cases:

1. In this case we know that $\alpha_s(s^*, c^*.addr, c\ell^*) \subseteq \alpha_{c^*}(S^{**}) = \alpha_{c^*}(S^*)$ for some $c\ell^* \in \mathbb{B}$. Since, we know that s^* is a call state (Lemma 17), we know that $s^* = (\mu^*, \iota^*, \sigma^*, \eta^*)$ for some μ^*, ι^*, σ^*, and η^* such that the conditions in Definition 17 are satisfied. Since s^* is a call state, $\iota^*.code[\mu^*.\mathsf{pc}] = \mathsf{CALL}$[1]. As s^* is strongly consistent with c^* (by Lemma 39), also $\iota^*.code = c^*.code$ and hence $\delta(c^*) \supseteq (\!|\mathsf{CALL}|\!)_{\mathsf{pc}^*}$. In particular, the second abstract CALL rule (C2) is applicable on $\alpha_s(s^*, c^*.addr, c\ell^*) \subseteq \alpha_{c^*}(S^{**})$ and hence one can derive $\mathsf{MState}_0((0, \lambda x.\,0), \lambda x.\,0, \sigma^*(c^*.addr).stor, 1)$. Additionally, we have that $\alpha_s((\mu', \iota', \sigma', \eta), c^*.addr, (S^* \neq \epsilon)) = \mathsf{MState}_0((0, \lambda x.\,0), \lambda x.\,0, \sigma'(c^*.addr).stor, 1)$ (since, μ' is an initial machine state and S^* is non-empty). Together with $s'.\sigma(c^*.addr).stor = s^*.\sigma(c^*.addr).stor$ and $\sigma(a).stor = \sigma'(a).stor$ for all a (since the call rule does not affect a contract's storage), we can conclude that $\alpha_s(s^*, c^*.addr, c\ell^*) \cup \delta(c^*) \vdash \alpha_s((\mu', \iota', \sigma', \eta), c^*.addr, 1)$. Due to the monotonicity of $\delta(c^*)$ (Theorem 7), we know that there is some $\Delta_i \geq \alpha_s((\mu', \iota', \sigma', \eta), c^*.addr, 1)$, such that $\Delta_{S^*} \cup \delta(c^*) \vdash \Delta_i$ (since $\Delta_{S^{**}} \geq \alpha(S^{**}) \supseteq \alpha_s(s^*, c^*.addr, c\ell^*)$. Consequently:

$$\begin{aligned}\Delta_{S^*} \cup \delta(c^*) \vdash \Delta_{S^*} \cup \Delta_i \\ \geq \alpha(S^*) \cup \alpha_s((\mu', \iota', \sigma', \eta), c^*.addr, 1) \\ = \alpha((\mu', \iota', \sigma', \eta)::S^*) \\ = \alpha(S')\end{aligned}$$

2. In this case, we get from the inductive hypothesis for m (since $m < n$) that there exists some Δ_H such that $\Delta_H \geq \alpha(HALT(\sigma^*, gas^*, d^*, \eta^*)_{c^*}::S^\dagger)$ and $\Delta_I \cup \delta(c^*) \vdash \Delta_H$, and additionally $|S^\dagger| > 0$. Consequently also

[1] This is a simplifying assumption made here. Actually $\iota^*.code[\mu^*.\mathsf{pc}] \in \{\mathsf{CALL}, \mathsf{STATICCALL}, \mathsf{CREATE}\}$. Since, the abstract semantics of these instructions have the same rules (up to minor differences in the preconditions of calling), exactly the same argumentation applies as shown here for the case of CALL.

$\Delta_H \geq \{\mathsf{Halt}(\sigma^*(c^*.addr).stor, 1)\} \cup \alpha(S^\dagger)$ Additionally we know that $S^{**} = S_1 +\!\!+ [s^1{}_{c^*}]$ for some S_1 and some s^1 from Lemma 24 (since the first state on top of S needs to be annotated with c^*). Additiofnally we can conclude from Lemma 17 that s^1 is a call state. From Lemma 39, we know that s^1 is strongly consistent with c^* and hence $s^1.\iota.code = c^*.code$. As s^1 is a call state, hence also $c^*.code[s^1.\mu.\mathsf{pc}] = \mathsf{CALL}$ and consequently $\delta(c^*) \supseteq (\!|\mathsf{CALL}|\!)_{s^1.\mu.\mathsf{pc}}$[2]. In addition we have that $\alpha_s(s^1, c^*.addr, 0) \leq \Delta_{S^{**}}$ and since s^1 is a call state all preconditions of rule C3 in $(\!|\mathsf{CALL}|\!)_{s^1.\mu.\mathsf{pc}}$ are satisfied. More precisely $\alpha_s(s^1, c^*.addr, (S^\dagger \neq \epsilon)) \cup \delta(c^*) \cup \{\mathsf{Halt}(\sigma(c^*.addr).stor, 1)\} \vdash \{\mathsf{MState}_0((0, \lambda x.\, 0), \lambda x.\, 0, \sigma^*(c^*.addr).stor, 1)\}$ (since $|S^\dagger| > 0$). By the monotonicity of $\delta(c^*)$ (Theorem 7) hence there is some Δ_x such that $\Delta_x \geq \{\mathsf{MState}_0((0, \lambda x.\, 0), \lambda x.\, 0, \sigma(c^*.addr).stor, 1)\}$ and $\Delta_H \cup \Delta_{S^*} \cup \delta(c^*) \vdash \Delta_x$. Since $\sigma^*(c^*.addr).stor = \sigma'(c^*.addr).stor$ and as $(\mu', \iota', \sigma', \eta)$ is an initial state we know that $\alpha_s((\mu', \iota', \sigma', \eta), c^*.addr, (S^* \neq \epsilon)) = \{\mathsf{MState}_0((0, \lambda x.\, 0), \lambda x.\, 0, \sigma^*(c^*.addr).stor, 1)\}$ which concludes the proof since

$$\begin{aligned}\Delta_{S^*} \cup \delta(c^*) &\vdash \Delta_{S^*} \cup \Delta_H \cup \delta(c^*)\\ &\vdash \Delta_{S^*} \cup \Delta_x\\ &\geq \alpha(S^*) \cup \alpha_s((\mu', \iota', \sigma', \eta), c^*.addr, (S^* \neq \epsilon))\\ &= \alpha(S')\end{aligned}$$

Halt (returning from regular halting). Then $s' = \mathit{HALT}(\sigma', gas', d', \eta')$, $S^{**} = s''{}_{c''} :: S^\dagger$ and $S' = s'''{}_{c''} :: S^\dagger$. We make a case distinction on $c'' = c^*$:

$c'' \neq c^*$ In this case, clearly, $\alpha(S^*) \supseteq \alpha(S^\dagger)$ and $\alpha(S') = \alpha(S^\dagger)$ and consequently $\Delta_{S'} \geq \alpha(S')$ and hence the claim trivially follows by the reflexivity of $\vdash$.

$c'' = c^*$ In this case $\alpha(S^*) \supseteq \alpha_s(s'', c^*.addr, c\ell'') \cup \alpha(S^\dagger)$ and $\alpha(S') = \alpha_s(s''', c^*.addr, c\ell'') \cup \alpha(S^\dagger)$. From Lemma 17, we know that s'' is a call state. With Lemma 39, we additionally have that $s''.\iota.code = c^*.code$ and hence also $c^*.code[s''.\mu.\mathsf{pc}] = \mathsf{CALL}$[3]. Consequently $\delta(c^*) \supseteq (\!|\mathsf{CALL}|\!)_{s''.\mu.\mathsf{pc}}$. In addition we have that $\alpha_s(s'', c^*.addr, c\ell'') \leq \Delta_{S^*}$ and since s'' is a call state, all preconditions of rule Equation C1 in $(\!|\mathsf{CALL}|\!)_{s''.\mu.\mathsf{pc}}$ are satisfied. More precisely $\alpha_s(s'', c^*.addr, c\ell'') \cup \delta(c^*) \vdash \mathsf{MState}_{s''.\mu.\mathsf{pc}+1}((|s''.\mu.\mathsf{s}| - 6, \mathsf{stackToArray}(s''.\mu.\mathsf{s})[|s''.\mu.\mathsf{s}| - 7 \to \top]), \lambda x.\, \top, \lambda x.\, \top) = p$. We know additionally that $\alpha_s(s''', c^*.addr, c\ell'') = \mathsf{MState}_{s'''.\mu.\mathsf{pc}}((|s'''.\mu.\mathsf{s}|, \mathsf{stackToArray}(s'''.\mu.\mathsf{s})), \mathsf{toWordMem}(s'''.\mu.m), s'''.\mu.\sigma(c^*.addr).stor)$ Since $s'''.\mu.\mathsf{pc} = s''.\mu.\mathsf{pc} + 1$, $|s'''.\mu.\mathsf{s}| = |s''.\mu.\mathsf{s}| - 6$ and for all $i \in \{0, \ldots, |s''.\mu.\mathsf{s}| - 8\}$ we have $s''.\mu.\mathsf{s}[i] = s''.\mu.\mathsf{s}[i]$, it holds that $p \geq \alpha_s(s''', c^*.addr, c\ell'')$ (since $\lambda x.\, \top$ is the top element for mappings $f \in \mathbb{N} \to \hat{D}$ and $\top \geq s'''.\mu.\mathsf{s}[0]$, cf. Lemma 33). So since $\alpha(S^*) \cup \delta(c^*) \vdash p$ there is by the monotonicity of $\delta(c^*)$ (Theorem 7) some Δ_p such that $\Delta_{S^*} \cup \delta c^* \vdash \Delta_p$

[2] See footnote 1

[3] See footnote 1

and $\Delta_p \geq p$. Consequently we can conclude the proof:

$$\begin{aligned}\Delta_{S^*} \cup \delta(c^*) \vdash \Delta_p \cup \Delta_{S^*} \\ &\geq \{p\} \cup \alpha(S^*) \\ &\geq \alpha_s(s''', c^*.addr, c\ell'') \cup \alpha(S^\dagger) \\ &= \alpha(S')\end{aligned}$$

The same arguments apply for returning from exceptional halting.

□

C.3 Checking Security Properties with *eThor*

In this section, we discuss how the security properties presented in subsection 4.2.5 are implemented in *eThor* using *HoRSt*. In particular, we explain how reachability properties can be abstracted as queries using the example of the call unreachability property. We then illustrate the infrastructure for proving functional correctness properties as well as the one for automated soundness and precision testing.

C.3.1 From Single-entrancy to Call Unreachability

We need to show that call unreachability is a sufficient criterion for showing single-entrancy.

Note that, in the following, we will implicitly assume that all executions are collision-resistant for $\{address\,(c)\}$.

Theorem 8. *If a contract $c \in \mathcal{C}$ is call unreachable and c's code does not contain DELEGATECALL and CALLCODE instruction then c is also single-entrant.*

Proof. Assume towards contradiction c not to be single-entrant. Then there exists a reachable configuration $(\Gamma, s_c{::}S)$ and states s, s' and a call stack S' such that $\Gamma \vDash s_c{::}S \rightarrow^* {s'}_c{::}S' +\!\!+ s_c{::}S$ (*) and a state s'' and a contract c' such that $\Gamma \vDash {s'}_c{::}S' +\!\!+ s_c{::}S \rightarrow^* {s''}_{c'}{::}{s'}_c{::}S' +\!\!+ s_c{::}S$ (**). We show that there is an initial execution state s^i that is strongly consistent with c, such that $\Gamma \vDash {s^i}_c{::}S \rightarrow^* {s'}_c{::}S' +\!\!+ s_c{::}S$ And $code\,(c)[s'.\mu.\mathsf{pc}] \in Inst_{call}$. By (*) and Lemma 20 we know that there exists a reachable initial execution state s^i such that $\Gamma \vDash {s^i}_c{::}S \rightarrow^* s_c{::}S$ and hence also $\Gamma \vDash {s^i}_c{::}S \rightarrow^* {s'}_c{::}S' +\!\!+ s_c{::}S$. Since each reachable configuration of a contract not containing DELEGATECALL and CALLCODE is strongly consistent with its annotation (Lemma 38), we know that also s^i is strongly consistent with c. By (*) and (**) we can conclude that ${s''}_{c'}{::}{s'}_c{::}S' +\!\!+ s_c{::}S$ is reachable and hence by Lemma 17 s' is a call state. By Lemma 15 we hence know that $s.\iota.\mathsf{code}[s'.\mu.\mathsf{pc}] \in Inst_{call}$ and since s' is strongly consistent with c, we can conclude that also $code\,(c)[s'.\mu.\mathsf{pc}] \in Inst_{call}$. □

C.3.2 From Reachability Properties to Queries

All reachability properties introduced in subsection 4.2.5 can be seen as instances of properties of the following form:

$$\mathcal{R}(P,R) := \forall s.\, P([s]) \implies \neg\exists S',\, \Gamma \vDash s_{c^*} :: S \rightarrow^* S' +\!\!+ S \;\wedge\; R(S')$$

where s is assumed to be strongly consistent with c^* and S' is assumed to be non-empty. We will refer to properties of this form in the following as *unreachability* properties.

For the sake of presentation, we will in the following interpret predicates P, R as the sets of elements satisfying these predicates. Additionally, we will overload the abstraction function α to operate on sets of configurations hence writing $\alpha_{c^*}(R)$ for $\bigcup_{S' \in R} \alpha_{c^*}(S')$ and $\alpha_{c^*}(P)$ for $\bigcup_{[s_{c^*}] \in P} \alpha_{c^*}([s_{c^*}])$.

Following Theorem 3 for proving such properties it is sufficient to give some set Δ_P such that $\Delta_P \geq \alpha_{c^*}(P)$ and to show, for any set Δ_r over-approximating $\alpha_{c^*}(r)$ for some $r \in R$ that $\Delta_P \nvdash \Delta_r$. Instead of showing this property for all possible sets Δ_r, it is sufficient to find a query set Δ_{query} so that every Δ_r fully contains at least one element in this set. Then one can conclude that not being able to derive any of the elements $Q \in \Delta_{query}$ then also one cannot derive any valid abstraction Δ_r of a configuration in R.

$$\forall r \in R.\, \Delta_r \geq \alpha_{c^*}(r) \implies \exists Q \in \Delta_{query}.\, Q \subseteq \Delta_r \tag{C.7}$$

Proving $\Delta_R, \delta(c^*) \nvdash Q$ for all $Q \in \Delta_{query}$ then implies that $\mathcal{R}(P,R)$ holds.

Usually, such a set can be easily constructed from R as follows:

$$\begin{aligned}\Delta_{query}(R) := \{P' \mid \exists S\, P.\, S \in R \;\wedge\; P = \alpha_{c^*}(S) \;\wedge\; P \leq P' \\ \wedge\; (\forall p' \in P'.\, \exists p \in P.\, p \leq_p p') \;\wedge\; (\forall p'\, q' \in P'.\, p' \leq_p q' \implies p = q)\}\end{aligned} \tag{C.8}$$

Intuitively, it is sufficient to query for the most concrete abstraction (as given by α^*_c) of the concrete configurations in R and all predicate-wise ($\leq_p$) coarser abstractions of those.

We formally state this property in the following lemma:

Lemma 49. *Let $S_s \subseteq \mathcal{C} \times \mathcal{C}$ be a small-step semantics and $(\mathcal{D}, \mathcal{S}, \alpha, \Lambda)$ a sound abstraction thereof. Further, let P, $R \subseteq \mathcal{C}$ be predicates on configurations and Δ_P be an abstract configuration s.t. $\Delta_P \geq \alpha(P)$. Then it holds that*

$$(\forall Q \in \Delta_{query}(R).\, \Delta_P, \Lambda \nvdash Q) \implies \mathcal{R}(P,R)$$

Proof. By contraposition. Assume that $\neg\mathcal{R}(P,R)$. Then $\exists c.P(c) \wedge \exists c'.\, (c,c') \in S_s \;\wedge\; R(c')$. Since $P(c)$ we have that $c \in P$ and hence $\alpha(c) \subseteq \alpha(P)$. So in particular $\Delta_P \geq \alpha(c)$. By Theorem 3 we know that $\Delta_P, \Lambda \vdash \Delta_{c'}$ such that $\Delta_{c'} \geq \alpha(c')$. By the construction of $\Delta_{query}(R)$ we know that there is some $Q \in \Delta_{query}(R)$ such that $Q \subseteq \Delta_{c'}$. Consequently since $\Delta_P, \Lambda \vdash \Delta_{c'}$, then also $\Delta_P, \Lambda \vdash Q$. □

Thus, it is generally sufficient to query for the reachability of all elements in $\Delta_{query}(R)$ in order to prove an unreachability property $\mathcal{R}(P, R)$. Note that by the definition of derivability, the empty set $\emptyset$ can always be derived from any abstract configuration and set of Horn clauses. Hence if $\alpha_{c^*}(S')$ is empty for some $S' \in R$ then by the non-reachability of the abstractions we cannot make any conclusions about $\mathcal{R}(P, R)$. This is also reflected in the lemma since if $\alpha_{c^*}(S')$ for some $S' \in R$ then also $\emptyset \in \Delta_{query}(R)$ and hence the non-derivability of all $Q \in \Delta_{query}(R)$ can never be shown.

Further, note that a (reachability) query in an SMT-solver is of the form (Φ, P) as defined in Appendix C.2.1, where Φ is a set of quantifier-free constraints over the variables in X and P is a set of predicate applications over variables in X. The SMT solver then checks whether there exists a variable assignment V such that $V \vDash \Phi$ and $\Delta, \Lambda \vdash V(P)$. In other words it checks that

$$\exists Q \in \{Q \mid \exists V.\, V \vDash \Phi \ \wedge \ Q = V(P)\}.\, \Delta, \Lambda \vdash Q$$

Hence to show that $\forall Q \in \Delta_{query}(R).\, \Delta, \Lambda \vdash Q$ it is not required to individually query all elements in $\Delta_{query}(R)$, but it is sufficient to partition $\Delta_{query}(R)$ into sets $\Delta_{query}(R) = \Delta_1 \cup \cdots \cup \Delta_n$ such that $\Delta_i = \{Q \mid \exists V.\, V \vDash \Phi_i \ \wedge \ Q = V(P_i)\}$ for some Φ_i and P_i. Then it is sufficient to query for the unreachability of all elements in Δ_i for all $0 < i \leq n$ what can be done with a single query for each Δ_i.

We will next show how this theoretical result can be used in practice and in particular at the level of *HoRSt*.

Initialization. For checking an unreachability property $\mathcal{R}(P, R)$, we need to show the non-derivability of a valid query set Δ_{query} from some abstract configuration $\Delta_P \geq \alpha(P)$. This requires to axiomatize such an abstract configuration Δ_P. This can be easily done in *HoRSt* by providing rules having `true` as a single premise. For axiomatizing that the execution starts in an initial machine state as required for the call unreachability property defined in Definition 10 we can add the following rule to the analysis specification:

```
1 rule initOp :=
2  clause true => MState{0}(0, [@V(0)], [@V(0)], [@T], false);
```

As the precondition P of the call unreachability property requires the top state s (that also serves as the zero-bar for the call level) to be initial, $\alpha(s)$ can contain only predicate applications of the form $\mathsf{MState}_0((0, \lambda x.\, 0), \lambda x.\, 0, m, 0)$ where m is some memory mapping. However, $\lambda x.\, \top$ (`[@T]` in *HoRSt*) over-approximates all memory arrays and hence $\Delta_P = \{\mathsf{MState}_0((0, \lambda x.\, 0), \lambda x.\, 0, \lambda x.\, \top, 0)\} \geq \alpha(P)$.

Queries. In addition to the syntax for writing an analysis specification, *HoRSt* also provides mechanisms for the interaction with the underlying SMT solver. More precisely it supports syntax for specifying *queries* and *tests*. Syntactically, queries consist of a list of premises (as in a clause). A query leads to the invocation of the SMT solver to test whether the conjunction of those premises is derivable from the given initialization using the specified rules. The query will result in `SAT` in case that all premises are derivable and in `UNSAT` in case that the conjunction of premises can be proven to be non-derivable.

```
query reentrancyCall
  for (!id: int) in ids(),
      (!pc:int) in pcsForIdAndOpcode(!id, CALL)
  [?sa: array<AbsDom>, ?mem: array<AbsDom>,
      ?stor: array<AbsDom>, ?size:int]
    MState{!id, !pc}(?size, ?sa, ?mem, ?stor, true);
```

Figure C.6: *HoRSt*-query for reentrancy.

In order to check for reachability of abstract configurations, *HoRSt* allows for the specification of (reachability) queries that can also be generated from selector functions. The query shown in Figure C.6 for instance checks for reentrancy by checking if any CALL instruction is reachable at call level 1[4]. It therefore is an implementation of the reachability property introduced in subsection 4.2.5. This query can be obtained from the call unreachability property defined in Definition 10 which is of the form $\mathcal{R}(P, R)$ with $R := \{s_{c^*} :: S' \mid |S'| > 0 \ \wedge \ c^*.code[s, \mu.\mathsf{pc}] \in Inst_{call}\}$. Intuitively, we can split this property into a set of different properties $\mathcal{R}(P, R_i)$ where i ranges over the set of CALL instructions in c^*. More precisely, let $R_i := \{s_{c^*} :: S' \mid |S'| > 0 \ \wedge \ s, \mu.\mathsf{pc} = i\}$ then it holds that

$$\mathcal{R}(P, R) \Leftrightarrow \forall i \in \{i \mid c^*.code[i] \in Inst_{call}\}. \mathcal{R}(P, R_i)$$

Then each instance of the `reentrancyCall` query specifies one query set Δ^i_{query} that satisfies Equation C.7 for R_i. Thus showing the underivability of all those sets from Δ_P proves the claim. Intuitively, Δ^i_{query} satisfies Equation C.7 for R_i because for each $r \in \Delta^i_{query}$, $\alpha^*_c(r)$ contains an application of a predicate MState_i with argument $c\ell = 1$ and for each possible abstraction of this predicate application, there is some $Q \in \Delta^i_{query}$ containing it.

C.3.3 Functional Correctness

For checking functional correctness, some modifications to the abstract semantics are necessary. This is because the different contract executions need to be bound to the corresponding input data of the call and to account for return data. We will in the following overview the relevant changes and motivate that similar modifications can easily be incorporated for reasoning about other dependencies with the execution or blockchain environment. We will present the relevant modifications in *HoRSt* syntax so that the explanations serve as a guide to the enhanced version of the semantics [ext20].

First, the relevant predicates need to be enriched with a corresponding representation of the call data. We decided to represent call data as a word array with the particularity that the array's first element represents only 4 bytes. This is due to the call conventions enforced by the *Solidity* compiler, which interpret the first 4 bytes of input data as the hash of the called function's signature to dispatch function calls properly. In addition to the call data, we introduce a new predicate representing the return data of a call.

Formally, we arrive at the following predicate definitions:

[4]There are corresponding queries for other relevant transaction initiating instructions.

```
1 rule opCallDataLoad :=
2  for (!id: int) in ids(), (!pc: int) in pcsForIdAndOpcode(!id, CALLDATALOAD), (!a:
       int) in argumentsOneForIdAndPc(!id, !pc)
3  clause [?x: AbsDom, ?size: int, ?sa: array<AbsDom>, ?mem: array<AbsDom>, ?stor:
       array<AbsDom>, ?cl: bool, ?p: int, ?v: AbsDom, ?cdata: CallData]
4   MState{!id, !pc}(?size, ?sa , ?mem, ?stor, ?cl, ?cdata), ?size > 0,
5   !a != ~1, // in case that the position could be pre-computed, use it for
        accessing the position more precisely
6   ?v = accessWordCalldata{!a}(?cdata) // accesses word at the corresponding
        position of the call data
7   => MState{!id, !pc +1}(?size, store ?sa (?size -1) (?v), ?mem, ?stor, ?cl, ?cdata
       ),
8  clause [?x: AbsDom, ?size: int, ?sa: array<AbsDom>, ?mem: array<AbsDom>, ?stor:
       array<AbsDom>, ?cl: bool, ?cdata: CallData, ?p: int, ?v: AbsDom]
9   MState{!id, !pc}(?size, ?sa , ?mem, ?stor, ?cl, ?cdata), ?size > 0,
10  !a = ~1, // if the argument could not be preecomputed, extract the argument from
        stack
11  ?x = select ?sa (?size - 1),
12  ?v = (isConcrete(?x)) ? (accessWordCalldataEven(extractConcrete(?x), ?cdata)) : (
       @T) // if the offset is concrete, try to access the word at the given
        position. This will only result in a concrete result if the value is a word
        position
13  => MState{!id, !pc +1}(?size, store ?sa (?size -1) (?v), ?mem, ?stor, ?cl, ?cdata
       );
```

Figure C.7: Rule for CALLDATALOAD in the enhanced abstract semantics.

```
1 datatype CallData := @D<int*array<AbsDom>>;
2 pred MState{int*int}: int * array<AbsDom> * array<AbsDom> * array<AbsDom> * bool *
      CallData;
3 pred Exc{int}: bool;
4 pred Halt{int}: array<AbsDom> * AbsDom * bool * CallData;
5 pred ReturnData{int}: int * AbsDom * bool * CallData;
```

Note that we represent call data as a pair of its size and an array of abstract words. Also, we added to the **Halt** predicate an argument representing the return data size. This argument is from the abstract domain with **@T** indicating that the concrete size of the return data is unknown. The **ReturnData** predicate maps the positions of the return data (word) array to the corresponding values that it holds.

The existing rules simply propagate the call data array with the only addition that the CALLDATALOAD instruction now accesses the call data array instead of over-approximating the loaded value. The new rule for CALLDATALOAD is depicted in Figure C.7. The CALLDATALOAD operation takes as an argument a value from the stack that specifies the position in the byte array where a word should be accessed. The rule is split into two clauses to benefit from the preanalysis. In case that the position of call data is known upfront, the call data array `?call` can be accessed more precisely. Since we model the call data as a word instead of a byte array (similar to our memory abstraction), either a word loaded from it consists of a full word in the word array or needs to be composed out of two neighboring words. Composing two integers (interpreting them as byte arrays), however, requires exponentiation as defined in the append function in subsection C.2.3. *z3* is not able to handle general exponentiation - for this reason,

```
1 rule opHaltOnReturn :=
2     for (!id: int) in ids(), (!pc: int) in pcsForIdAndOpcode(!id, RETURN)
3     let
4       macro #StackSizeCheck := MState{!id,!pc}(?size, ?sa, ?mem, ?stor, ?cl, ?cdata
            ), ?size > 1
5     in
6     clause [?sa: array<AbsDom>, ?mem: array<AbsDom>, ?stor: array<AbsDom>, ?size:
          int, ?cl: bool, ?cdata: CallData, ?length: AbsDom]
7         #StackSizeCheck, ?length = select ?sa (?size-2)
8         => Halt{!id}(?stor, ?length, ?cl, ?cdata),
9     clause [?sa: array<AbsDom>, ?mem: array<AbsDom>, ?stor: array<AbsDom>, ?size:
          int, ?cl: bool, ?offset: AbsDom, ?length: AbsDom, ?o: int, ?l:int, ?p:int,
          ?v: AbsDom, ?cdata: CallData]
10        #StackSizeCheck, ?offset = select ?sa (?size-1), ?length = select ?sa (?
              size-2), // select top values on the stack
11        isConcrete(?offset), isConcrete(?length),
12        ?o = extractConcrete(?offset), ?l = extractConcrete(?length),
13        ?p >= 0, (?p * 32) < ?l, // write all words that are still within the
              length
14        ?v = accessWordMemoryEven(?o + ?p, ?mem)
15        => ReturnData{!id}(?p, ?v, ?cl, ?cdata), // careful: the Return data
              predicate is also inhabited in words!
16    clause [?sa: array<AbsDom>, ?mem: array<AbsDom>, ?stor: array<AbsDom>, ?size:
          int, ?cl: bool, ?offset: AbsDom, ?length: AbsDom, ?o: int, ?l:int, ?p:int,
          ?v: AbsDom, ?cdata: CallData]
17        #StackSizeCheck, ?offset = select ?sa (?size-1), ?length = select ?sa (?
              size-2), // select top values on the stack
18        ~isConcrete(?offset), isConcrete(?length), // knowing only the length, we
              write top at the places in the specified range
19        ?l = extractConcrete(?length), ?p >= 0, ?p * 32 < ?l
20        => ReturnData{!id}(?p, @T, ?cl, ?cdata),
21    clause [?sa: array<AbsDom>, ?mem: array<AbsDom>, ?stor: array<AbsDom>, ?size:
          int, ?cl: bool, ?offset: AbsDom, ?length: AbsDom, ?o: int, ?l:int, ?p:int,
          ?v: AbsDom, ?cdata: CallData]
22        #StackSizeCheck, ?length = select ?sa (?size-2), ~isConcrete(?length), ?p
              >= 0
23        => ReturnData{!id}(?p, @T, ?cl, ?cdata);
```

Figure C.8: Rule for RETURN in the enhanced abstract semantics.

we can only compute such exponentiations (by unfolding to multiplications) whose exponent is known upfront. Thus, the first rule in Figure C.7 handles the case where the argument to the call is known upfront: the `accessWordCallData` function expects the position as a parameter and computes the accessed word precisely from the call data array by unrolling exponentiation. The second rule handles the case where the argument to the call is not known upfront. In case that during the analysis, it can be detected to be concrete (by the function `isConcrete`), the `accessWordCalldataEven` function is used to access the call data at the corresponding position. This function, however, only yields a precise result in the case that the provided position corresponds to the beginning of a word in the call data array. Otherwise, it over-approximates the result as `T`.

The **`ReturnData`** predicate is inhabited by the rules that model regular halting. We exemplarily show the rule of the RETURN opcode depicted in Figure C.8. The RETURN instruction in EVM reads the memory offset and the return data length from the stack and returns the corresponding

memory fragment as a byte array. In our abstraction, the return data is modeled by an own predicate that holds words instead of bytes. This design choice follows the one made for the word-indexed memory and the call data array, which hold words instead of bytes as well for performance reasons. The RETURN semantics is closely reflected in the abstract RETURN rule: the first clause of the rule inhabits the **`Halt`** predicate, reading the size of the return data from the stack. The next three clauses inhabit the **`ReturnData`** predicate, differentiating depending on how much information on the return data (size and memory offset) are known: If both memory offset and length of the data are known, for each word position `?p` the corresponding memory word is read from the memory array `?mem` (using the function `accesswordMemoryEven`) and written into the **`ReturnData`** predicate. The next clause describes the case where the memory offset is unknown, but the size of the return data is known. In this case, we cannot know which (concrete values) form the return data but can only approximate all possible return data words (as determined by the size of the array) with **`@T`**. The last clause covers the case where the length of the return data is unknown. Since it is unclear in this case whether data should be returned in the first place (since the length could be 0), all potential positions of the return data array are over-approximated by **`@T`**.

Finally, the functional correctness queries for the addition function of the SafeMath library are depicted in Figure C.9. We first specify the call data for a call to the `add` function of the SafeMath library as an operation `callAdd` returning an **`CallData`** element when being provided with the arguments to the call. Since the `add` function expects 2 integer arguments the `callAdd` function returns call data of size 68 $(4 + 2 * 32)$ bytes where the **`@T`** array is initialized with the hash of the corresponding function signature as first element (which represents the first 4 bytes of the call data) and the arguments `x` and `y` as following two elements. Note that the hash of the function signature is provided by *Solidity* compilers via the so-called Ethereum Contract ABI (Contract Application Binary Interface). We plan to automatically generate an infrastructure for functional correctness queries on *Solidity* contracts from the contract ABI. The first functional correctness test `addOverflowNoHalt` requires that it is impossible to reach a **`Halt`** state (indicating regular halting) from a call to the `add` function in case that the addition of `?x` and `?y` overflows.

The second functional correctness test (`addNoOverflowCorrect`) checks whether it is possible (in case that no overflow occurs) to compute the expected result (or an over-approximation thereof) in the first place. Here `abseq` is the function implementing an equality test on the abstract domain, hence considering every concrete element to be potentially equal to **`@T`**. By the soundness of the analysis, if this query would turn out to be unsatisfiable, it would be impossible for the function to produce the correct result under any circumstances. This query, of course, does not prove that the function will always provide a result: This indeed is and should not be provable since any smart contract can always exceptionally halt when running out of gas. This test case serves as a sanity check that only becomes meaningful in conjunction with the following tests. The third and fourth functional correctness tests (`addNoOverflowHalt` and `addNoOverflowUnique`) prove that given non-overflowing arguments, if the function execution halts successfully, nothing but the correct result can be produced. In other words, it is impossible to halt successfully without producing the correct result. This property is composed

```
op callAdd(x: int, y: int): CallData :=
 @D(68, store (store (store [@T] 0 @V(1997931255)) 1 (@V(x))) 2 (@V(y)));

test addOverflowNoHalt expect UNSAT
  for (!id: int) in ids()
  [?x:int, ?y: int, ?z:int, ?p:int, ?stor: array<AbsDom>, ?rdsize:AbsDom]
   ?x >= 0, ?y >= 0, ?x < MAX, ?y < MAX, ?x + ?y >= MAX,
   Halt{!id}(?stor, ?rdsize, false, callAdd(?x, ?y));

test addNoOverflowCorrect expect SAT
  for (!id: int) in ids()
  [?res: AbsDom, ?x:int, ?y: int, ?z:int, ?rdsize:AbsDom, ?stor: array<AbsDom>]
   ?x >= 0, ?y >= 0, ?x + ?y < MAX,
   ReturnData{!id}(0, ?res, false, callAdd(?x, ?y)),
   Halt{!id}(?stor, ?rdsize, false, callAdd(?x, ?y)),
   abseq(?rdsize, @V(32)), abseq(?res, @V(?x + ?y));

test addNoOverflowHalt expect UNSAT
  for (!id: int) in ids()
  [?res: AbsDom, ?x:int, ?y: int, ?z:int, ?rdsize: AbsDom, ?stor: array<AbsDom>]
  ?x >= 0, ?y >= 0, ?x + ?y < MAX,
  Halt{!id}(?stor, ?rdsize, false, callAdd(?x, ?y)),
  ?rdsize != @V(32);

test addNoOverflowUnique expect UNSAT
  for (!id: int) in ids()
  [?res: AbsDom, ?x:int, ?y: int, ?z:int, ?rdsize: AbsDom, ?stor: array<AbsDom>]
    ?x >= 0, ?y >= 0, ?x + ?y < MAX,
    ReturnData{!id}(0, ?res, false, callAdd(?x, ?y)),
    ?res != @V(?x + ?y);
```

Figure C.9: Correctness queries for SafeMath's `add` function

of two queries since it needs to be shown that 1) It is impossible for the function to halt without returning a result of length 32 (corresponding to one word) as recorded in the **`Halt`** predicate and 2) It is impossible that the actual return value (as recorded in the **`ReturnData`** predicate) differs from the sum of the arguments.

The functional correctness tests for the other functions of the SafeMath library follow the same pattern.

C.3.4 Automated Testing in *HoRSt*

The setup for automated testing (see subsection 4.4.3) shown in Figure C.10 presents a use case for the Hoare-Logic-style reasoning capabilities of *eThor*.

Test cases in the official test suite come in two flavors: the first group consists of 490 test cases specifying a storage configuration as postcondition, the second group, consisting of 108 test cases, lacks a postcondition (which we interpret as exceptional halting).

To account for this test structure we declare four additional selector functions: The selector functions `preStorageForId` and `postStorageForId` provide tuples of storage offsets and values which specify the storage contents before and af-

```
1 sel preStorageForId: int -> [int*int];
2 sel postStorageForId: int -> [int*int];
3 sel emptyListIfNoPostConditionForId: int -> [bool];
4 sel dummyListIfNoPostConditionForId: int -> [bool];
5
6 rule initOp :=
7   for (!id:int) in ids()
8   clause
9     true
10    => MState{!id, 0}(0, [@V(0)], [@V(0)],
11      for (!offset: int, !value:int) in preStorageForId (!id):
12       x: array<AbsDom> -> store x !offset @V(!value), [@V(0)],
13      false);
14
15 test correctValues expect SAT
16   for (!id: int) in ids(),
17       (!b: bool) in emptyListIfNoPostConditionForId(!id)
18   [?stor: array<AbsDom>, ?i: int]
19    for (!offset: int, !value:int) in postStorageForId(!id):
20     && abseq(select ?stor !offset,@V(!value)),
21    (for (!offset: int, !value:int) in postStorageForId(!id):
22     || ?i = !offset) ? (true) : (abseq(select ?stor ?i,@V(0))),
23    Halt{!id}(?stor, false);
24
25 test uniqueValues expect UNSAT
26   for (!id: int) in ids(),
27       (!b: bool) in emptyListIfNoPostConditionForId(!id)
28   [?stor: array<AbsDom>]
29    for (!offset: int, !value:int) in postStorageForId(!id):
30     || absneq(select ?stor !offset,@V(!value)),
31    Halt{!id}(?stor, false);
32
33 test irregularHalt expect UNSAT
34   for (!id: int) in ids(),
35        (!b: bool) in dummyListIfNoPostConditionForId(!id)
36   [?stor: array<AbsDom>]
37    Halt{!id}(?stor, false);
```

Figure C.10: Setup for automated testing.

ter the execution of the contract. `emptyListIfNoPostConditionForId` and `dummyListIfNoPostConditionForId` generate an empty list, respectively a list with one element, depending on if there is a postcondition specified or not. Since rules are generated for the cross product of their selector functions return values, we can use these functions to generate different rules for different test cases while still using the same *HoRSt* inputs.

The rule for initialization, `initOp`, slightly differs from the definition used in the other experiments. The analysis is initialized to start in a storage as specified by `preStorageForId`.

To check for the reachability of a certain storage configuration, we generate the two queries `correctValues` and `uniqueValues`. `correctValues` is successful, if a **`Halt`** predicate is reachable whose storage contains 1) values abstractly equal to the values returned by `postStorageForId` at the offsets returned by `postStorage` and 2) a value abstractly equal to 0 for all offsets not returned by `postStorageForId`. `uniqueValues` is successful, if no

Halt predicate is reachable whose storage contains any value abstractly unequal to the values returned by `postStorageForId`. This is only the case, if every value in the memory is concrete. In sum, such a test case is considered to be solved *correctly* if `correctValues` is successful and considered to be solved *precisely* if `correctValues` and `uniqueValues` are successful. To check for exceptional halting, we just query for the unreachability of a regular **Halt** predicate (see `irregularHalt`). Such a query is considered solved precisely on success and imprecisely on failure, since reaching additional program states (**Halt** in this instance), which are not reachable in the concrete execution, indicates over-approximation.

APPENDIX D

Appendix to Chapter 5

D.1 Wormhole Attack

In this section, we first generically describe the wormhole attack. We then formally prove that no two-round multi-hop payment in a payment-channel network (without broadcasts) is robust against this attack.

First, we give a model for such two-round multi-hop payments that is capturing existing two-round constructions such as the standard HTLC-based construction as it is presented in [MMSK+17]. As we are only interested in the underlying locking mechanism of PCNs, we omit information on channel capacities and fees and simply model PCNs as graphs of the form $\mathbb{G} = (\mathbb{V}, \mathbb{E})$.

For arguing about the communication in protocols, we formally state the notions of a path in a PCN as well as of a communication round along a path.

Definition 4 (Path in a PCN)**.** *Let $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ be a PCN. We call a vector $(u_1, \ldots, u_n)$ of users a* path in $\mathbb{G}$ *if it holds for all $i \in [1, n-1]$ that $(u_i, u_{i+1}) \in \mathbb{E}$*

In the following, we represent a message from u_1 to u_2 with content m as a tuple $M = \langle u_1, m, u_2 \rangle$. Intuitively, one round of communication in a PCN allows for the consecutive execution of pairwise protocols along a path. This is formally captured by the following definition:

Definition 5 (Communication Round in PCNs)**.** *Let $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ be a PCN and let $\pi = (u_1, \ldots, u_n)$ be a path in $\mathbb{G}$ with length longer than one. We we call a vector of consecutive messages $(M_1, \ldots, M_l)$ a* round of communication along π *if the following conditions hold:*

1. *$M_1 = \langle u_1, m, u_2 \rangle$ for some message m*
2. *for all $i \in [1; l]$ it holds that either*

a) $M_i = \langle u_j, m, u_{j+1} \rangle$ *or*

b) $M_i = \langle u_{j+1}, m, u_j \rangle$ *for some* $j \in [1; n-1]$ *and some message* m

3. *for all* $i \in [1; l-1]$ *it holds that if* $M_i = \langle u_j, m_i, u_k \rangle$ *then either*

a) $M_{i+1} = \langle u_k, m, u_j \rangle$ *or*

b) $M_{i+1} = \langle u_{\max(j,k)}, m, u_{\max(j,k)+1} \rangle$ *for some message* m

Here condition 1) ensures that a round always starts with a message of the first user in the path to its right neighbor. Condition 2) makes sure that messages can only be exchanged between neighbors in the path. Finally, condition 3) encodes that every message in a path can either be followed by a message in the reversed direction (in case that a protocol between two neighbors is performed) or alternatively a protocol between the next two neighboring nodes is initiated (by the ‚right' party in the former protocol now sending a message to its right neighbor). Together these definitions ensure that only pairwise protocols can be performed, and once that, they need to be carried out consecutively along the path.

For describing the essence of the wormhole attack, we use an abstract view on payments in PCNs where we assume payments along a path π to consist of a *commitment phase* and a *releasing phase*. In the commitment phase (which constitutes a communication round along π), pairwise locks (e.g., HTLCs) between the parties along the payment path are created in pairwise locking protocols between the neighboring nodes.

Definition 6 (Commitment Phase in a PCN)**.** *Let* $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ *be a PCN and let* $\pi = (u_1, \ldots, u_n)$ *be a path in* $\mathbb{G}$*. A protocol encompassing one communication round along* π *is called a* commitment phase along π *if as a result each user* u_i *(for* $i \in [1; n]$*) learns some (shared) pieces of commitment information* $\ell_{i,i+1}$*,* $\ell_{i-1,i}$ *(with* $\ell_{n,n+1}$ *and* $\ell_{0,1}$ *being empty). We call* $\{\ell_{i,i+1}\}_{i \in 1,n-1}$ *the output of the commitment phase.*

In the releasing phase, starting from the payment's receiver, keys for 'opening' the locks are released (as the condition R in the lightning network). The releasing phase thereby consists of a communication round along *rev*(π) (the reversal of π). These keys should satisfy the property that each path node can – given a valid key for an outgoing lock – derive a key for its incoming lock.

Definition 7 (Releasing Phase in a PCN)**.** *Let* $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ *be a PCN and let* $\pi = (u_1, \ldots, u_n)$ *be a path in* $\mathbb{G}$*. Further let* C *be a commitment phase along* π *and* Vf *be a boolean function taking two arguments. Additionally, let* π' *be a subpath of* π*. A protocol encompassing one communication round along rev*(π') *is called a* releasing phase for C along *rev*(π') *if as a result, each user* u_i *in* π' *learns some information* $k_{i-1,i}$ *such that* $\mathsf{Vf}(\ell_{i-1,i}, k_{i-1,i}) = 1$.

With the notion of a subpath being defined as follows:

Definition 8 (Subpath)**.** *Let $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ be a PCN and let $\pi = (u_1, \ldots, u_n)$ be a path in $\mathbb{G}$. A path $\pi' = (u_1, \ldots, u_m)$ is considered a subpath of π if for all u_i in π' there are paths π_i (for $i \in [0; m]$) such that $\pi = \pi_0 \cdot (u_1) \cdot \pi_1 \cdot (u_2) \cdot \pi_2 \ldots \cdot \pi_{m-1} \cdot (u_m) \cdot \pi_m$ (where $\cdot$ denotes path concatenation).*

Note that we assume communication to be restricted to nodes connected by a direct link in the PCN (as ensured by Definition 5). This prevents that besides the specified messages in the releasing phase, keys can be sent to previous nodes in the path (e.g., via broadcast).

Figure D.1 shows the payment from Alice to Edward in the abstract setting. Initially, Edwards gives a trapdoor t to Alice. Using this, Alice starts the commitment phase by creating the lock $\ell_{A,B}$ with Bob. To this end, Alice and Bob might use their secret local states s_A and s_B. In the same fashion, all following pairwise locks are created in the commitment phase till reaching Edward. Edward then starts the releasing phase by using the trapdoor he initially sent to Alice for creating the key $k_{D,E}$ for opening lock $\ell_{D,E}$. From this key (and the information learned in the commitment phase), Dave can derive key $k_{C,D}$. In this fashion, the whole lock chain can be released.

Note that in the setting of only two rounds of communication (one along the payment path π and one along the reversed payment path $rev(\pi)$), the initial secret local states of the users involved in a payment are completely independent from π and consequently from the local states of the other nodes in the path. This is as none of the users received any path-specific information upfront.

As a consequence, two nodes u_i and u_j (with $1 < i + 1 < j$) on a payment path can exclude intermediate nodes u_k (with $i < k < j$) from taking part in the releasing phase as follows: After completing the commitment phase in an honest fashion, the releasing phase proceeds honestly till reaching u_j. At this point u_j can derive a key $k_{j-1,j}$ for releasing the lock $\ell_{j-1,j}$ with (honest) user u_{j-1}. Instead of releasing this lock, u_j forwards $k_{j-1,j}$ to u_i which again can use this key for producing a valid key for lock $\ell_{i-1,i}$ with its predecessor u_{i-1}. This is possible as no secret information from the nodes $\{u_k\}_{i<k<j}$ is required for generating a valid key for $l_{i-1,i}$. Otherwise also opening the final lock would already require secret information from some intermediate nodes. As these pieces of secret information from the intermediate nodes are, however, completely unrelated to the path and consequently from the trapdoor t, even the receiver could not earn the necessary knowledge from the trapdoor for opening the last lock. So finally, node u_i can

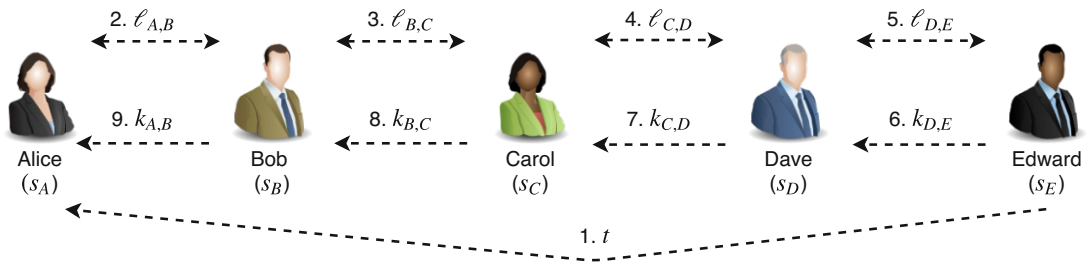

Figure D.1: Illustration of the abstract locking mechanism underlying payments in PCNs

release lock $\ell_{i-1,i}$ and consecutively all remaining locks can be released without contacting nodes $\{u_k\}_{i<k<j}$ at all. Together with the assumption that nodes $\{u_k\}_{i<k<j}$ cannot receive information through other channels than the direct communication with their immediate neighbors in the path and the fact that keys for locks can only be derived from the initial key, there is no way for nodes $\{u_k\}_{i<k<j}$ to open the locks with their successors in the path.

D.1.1 Inevitability of Wormhole Attacks in Two-round Payment Protocols

In PCNS, payments that only encompass two rounds of communication are inevitably vulnerable to wormhole attacks. [1] More specifically, this means that when no path-specific information was communicated to the intermediate nodes of the payment path before performing the payment, nodes located between two corrupted users in the path can always be bypassed in the releasing phase. This situation occurs in cases where the path is not known upfront, but routing is performed dynamically (e.g., [RMKG18]).

We formally prove the following theorem:

Theorem 6. *Let $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ be a PCN and let $\pi = (u_1, \ldots, u_n)$ be a path in $\mathbb{G}$ with length longer than two. Further, let u_i, u_j be nodes in π for some $0 < i < j \leq n$ and let $\pi' = (u_1, \ldots, u_i, u_j, \ldots, u_n)$ be the path omitting the nodes between u_i and u_j. Assume that u_1 and u_n share initial common knowledge t while all other nodes do not have any initial shared knowledge. Then each payment along π that is carried out by a protocol P consisting of a commitment phase C along π and a releasing phase along $rev(\pi)$ for C, can also be carried out by a protocol P' consisting of C and a releasing phase for C along $rev(\pi')$ given that u_i and u_j collude. Additionally, for all users u_k with $i < k < j$, P' is indistinguishable from the protocol only consisting of C.*

Proof. Let $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ be a PCN and let $\pi = (u_1, \ldots, u_n)$ be a path in $\mathbb{G}$ with length longer than two. Further, let u_i, u_j be nodes in π for some $0 < i < j \leq n$ and let $\pi' = (u_1, \ldots, u_i, u_j, \ldots, u_n)$ be the path omitting the nodes between U_i and U_j. Assume a payment along the path π with u_1 being the sender and u_n being the receiver. Without loss of generality, assume u_i and u_j to be controlled by the attacker and all other nodes on the path to be honest. We show that the view of honest nodes u_l with $l < i$ or $l > j$ in the scenario of u_i and u_j performing a wormhole attack on a successful payment don't differ from the view in the scenario of a successful payment. More precisely, we show how to construct a one-round successful releasing phase along $rev(\pi')$, where successful means that each user in π' can derive a valid key for its outgoing locks.

In addition, we show that the view of honest nodes u_m with $i < m < j$ in the scenario of the wormhole attack do not differ from their view in the scenario of an unsuccessful payment. More formally, we show that it is indistinguishable for those nodes whether the one-round releasing phase was omitted or carried out along $rev(\pi')$.

[1] Note that we assume here PCNs of the previously described structure hence requiring that payments encompass a commitment and a revealing phase and communication to be restricted to direct neighbors.

To this end, we first show how an attacker can simulate the behavior of the nodes $u_{i+1}, \ldots, u_{j-1}$ without changing the view of the honest nodes u_l with $l < i$ or $l > j$.

In the commitment phase, the locks along the path π have been created correctly. In the locking protocol between u_l and u_i, u_i behaves honestly, and consequently, u_l's view is the same as in the honest case. In parallel to starting the locking protocol between u_i and u_{i+1}, the attacker locally simulates the locking protocols for the user's $u_{i+1}, \ldots, u_j$ and creates simulated locks $\ell_{i+1}, \ldots, \ell_j$. This is possible as by sampling random local states for those nodes, the attacker can run the locking protocol locally. Finally, u_j can continue the commitment phase in an honest manner using as own local state, the one resulting from the simulated commitments. This cannot be distinguished by node u_{j+1} as its own local state is unrelated to the local states of the other intermediate nodes.

As we consider the case of a successful payment, the releasing phase will be performed honestly by nodes $u_n, \ldots u_{j+1}$. When u_{j+1} releases the lock between u_j and u_{j+1} with key k_j, then the attacker can simulate releasing the locks $\ell_{j-1}, \ldots, \ell_i$ locally without publishing the corresponding keys. This is possible as the attacker can use the local states of the intermediate nodes $u_{i+1}, \ldots, u_{j-1}$ from the simulated commitment phase for deriving keys $k_{j-1}, \ldots, k_{i-1}$. As k_{i-1} is hence also a valid key for the honestly created lock l_{i-1}, the releasing phase can from this point be concluded in an honest manner.

Finally, we can observe that nodes $u_{i+1}, \ldots, u_{j-1}$ are not contacted at all in the releasing phase of the payment, which is the same as in the case that the payment was unsuccessful, i.e., the releasing phase was not initiated by the receiver at all. □

D.2 AMHLs Correctness

In this section, we define the notion of correctness for AMHLs.

Definition 9 (Correctness of AMHLs)**.** *Let $\mathbb{L}$ be a AMHL, $\lambda \in \mathbb{N}^+$ and $n \in \mathsf{poly}(\lambda)$. Let $(U_0, \ldots, U_n) \in \mathbb{U}^n$ be a vector of users, $(\mathsf{sk}_0, \ldots, \mathsf{sk}_{n-1})$ and $(\mathsf{sk}_1^*, \ldots, \mathsf{sk}_n^*)$ two vectors of private keys and $(\mathsf{pk}_0, \ldots, \mathsf{pk}_{n-1})$ a vector of shared public keys such that for all $0 \leq i < n$, it holds that*

$$\{(\mathsf{sk}_i, \mathsf{pk}_i), (\mathsf{sk}_{i+1}^*, \mathsf{pk}_i)\} \leftarrow \langle \mathsf{KGen}_{U_i}(1^\lambda), \mathsf{KGen}_{U_{i+1}}(1^\lambda) \rangle.$$

Let $(s_0^I, \ldots, s_n^I)$ be vector of initial states and k_n be a key such that for all $0 \leq i < n$

$$\{s_0^I, \ldots, (s_n^I, k_n)\} \leftarrow \left\langle \begin{array}{c} \mathsf{Setup}_{U_0}(1^\lambda, U_1, \ldots, U_n) \\ \ldots \\ \mathsf{Setup}_{U_n}(1^\lambda) \end{array} \right\rangle$$

Furthermore, let $(\ell_0, \ldots, \ell_{n-1})$ be a vector of locks, $(s_1^L, \ldots, s_n^L)$ and $(s_0^R, \ldots, s_{n-1}^R)$ vectors of states, and $(k_0, \ldots, k_{n-1})$ a vector of keys such that for all $0 \leq i < n$, it holds that

$$\{(\ell_i, s_i^R), (\ell_i, s_{i+1}^L)\} \leftarrow \left\langle \begin{array}{c} \mathsf{Lock}_{U_i}(s_i^I, \mathsf{sk}_i, \mathsf{pk}_i) \\ \mathsf{Lock}_{U_{i+1}}(s_{i+1}^I, \mathsf{sk}_{i+1}^*, \mathsf{pk}_i) \end{array} \right\rangle$$

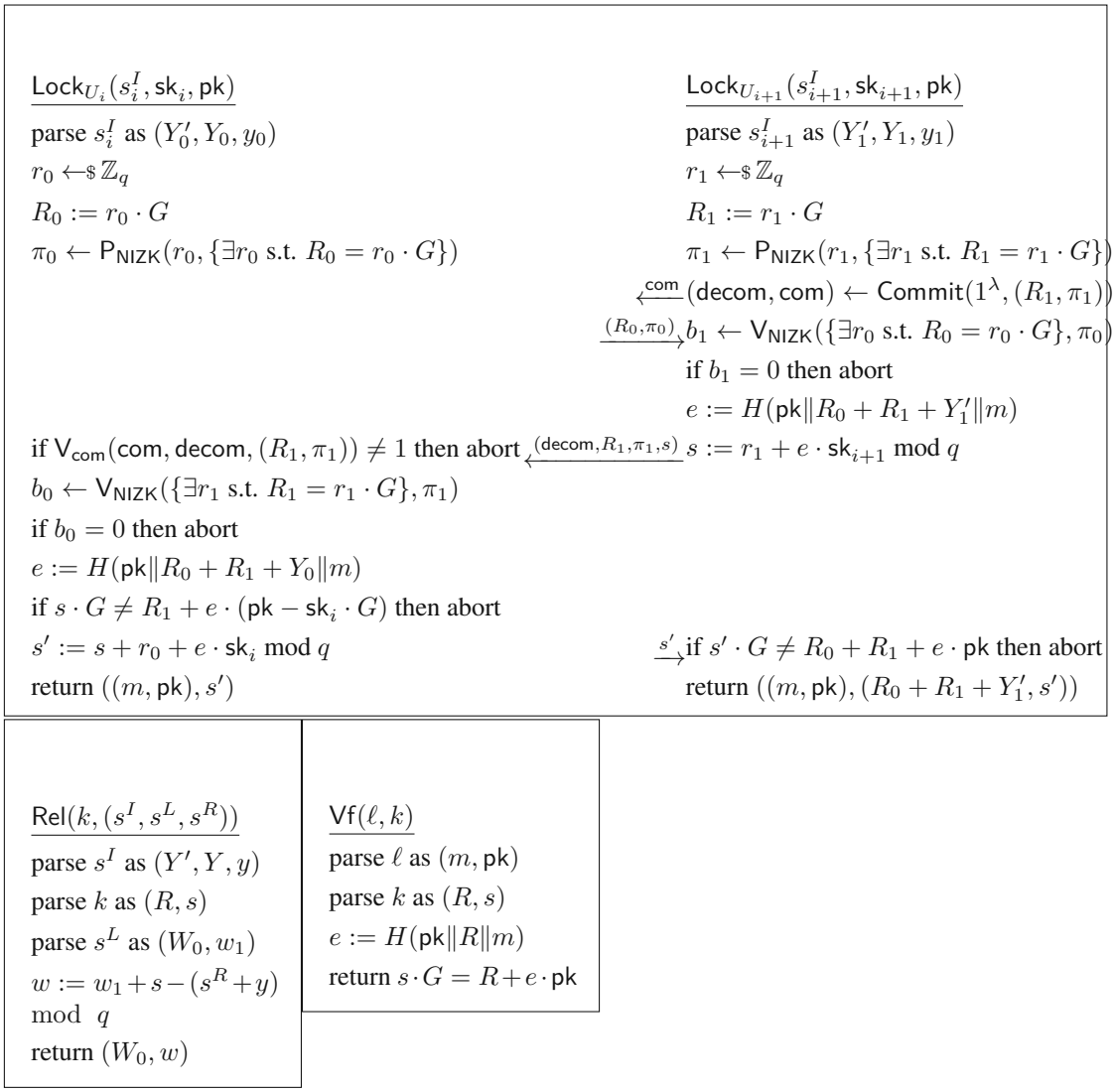

Figure D.2: Algorithms and protocols for the Schnorr-based construction. The Setup protocol is as defined in Fig. 5.5.

and

$$k_i \leftarrow \mathsf{Rel}(k_{i+1}, (s_{i+1}^I, s_{i+1}^L, s_{i+1}^R))$$

where s_n^R *is* $\perp$*. We say that* $\mathbb{L}$ *is correct if there exists a negligible function* negl *such that for all* $0 \leq i < n$ *it holds that*

$$\Pr\left[\mathsf{Vf}(\ell_i, k_i) = 1\right] \geq 1 - \mathsf{negl}(\lambda).$$

D.3 Schnorr-based Scriptless Construction

In the following, we cast the idea of Poelstra [Poea] in our framework.

Schnorr Signatures. Let $\mathbb{G}$ be an elliptic curve group of order q with base point G and let $H : \{0,1\}^* \to \{0,1\}^{|q|}$ be a collision resistant hash function (modeled as a random oracle). The key generation algorithm $\mathsf{KGen}_{\mathsf{schnorr}}(1^\lambda)$ of a Schnorr signature [Sch91] samples some $x \leftarrow\!\!\$\, \mathbb{Z}_q$ and sets the corresponding public key as $Q := x \cdot G$. To sign a message m, the signing algorithm $\mathsf{Sig}_{\mathsf{schnorr}}(\mathsf{sk}, m)$ samples some $k \leftarrow\!\!\$\, \mathbb{Z}_q$, computes $e := H(Q \| k \cdot G \| m)$, sets $s := k - xe$, and returns $\sigma := (R, s)$, where $R := k \cdot G$. The verification $\mathsf{Vf}_{\mathsf{schnorr}}(\mathsf{pk}, \sigma, m)$ returns 1 if and only if $s \cdot G = R + H(Q \| R \| m) \cdot Q$. Schnorr signatures are known to be strongly unforgeable against the discrete logarithm assumption [GMR88]. We assume the existence of a 2-party protocol $\Pi^{\mathsf{schnorr}}_{\mathsf{KGen}}$ where the two players, on input x_0 and x_1, set a shared public key $Q := (x_0 + x_1) \cdot G$. Such a protocol can be implemented using standard techniques and we denote the corresponding ideal functionality by $\mathcal{F}^{\mathsf{schnorr}}_{\mathsf{kgen}}$.

Description. Let $\mathbb{G}$ be an elliptic curve group of order q with base point G and let $H : \{0,1\}^* \to \{0,1\}^{|q|}$ be a hash function. The Schnorr-based construction is formally described in Fig. D.2. The key generation algorithm consists of a call to the $\mathcal{F}^{\mathsf{schnorr}}_{\mathsf{kgen}}$ functionality. At the end of a successful run, U_i receives (x_i, pk) whereas U_j obtains (x_j, pk), where $\mathsf{pk} := (x_i + x_j) \cdot G$. The setup of a AMHL is identical to the ECDSA-based construction and can be found in Fig. 5.5.

Prior to the locking phase, two users U_i and U_{i+1} (implicitly) agree on the value Y_i and on a message m to be signed. Each message is assumed to be unique for each session (e.g., contains a transaction identifier). The locking algorithm consists of an "incomplete" distributed signing of m. First, the two parties agree on a randomly chosen element $R_0 + R_1$ using a standard coin tossing protocol, then they set the randomness of the signature to be $R := R_0 + R_1 + Y_i$. Note that at this point the parties cannot complete the signature since they do not know the discrete logarithm of Y_i. Instead, they jointly compute the value $s := r_0 + r_1 + e \cdot (x_0 + x_1)$ as if Y_i was not part of the randomness, where e is the hash of the transcript so far. The resulting (R, s) is *not* a valid signature on m, since the additive term y^* (where $y^* \cdot G = Y_i$) is missing from the computation of s. However, rearranging the terms, we have that $(R, s + y^*)$ is a valid signature on m. This implies that, once the discrete logarithm of Y_i is revealed, a valid signature m can be computed by U_{i+1}. Leveraging this observation, U_{i+1} can enforce an *atomic* opening: The subsequent locking (between U_{i+1} and U_{i+2}) is conditioned on some $Y_{i+1} = Y_i + y_{i+1} \cdot G$. This way, the opening of the right lock reveals the value $y^* + y_{i+1}$ and U_{i+1} can immediately extract y^* and open its left lock with a valid signature on m. The security of the construction is shown by the following theorem. We refer the reader to Appendix D.5 for a full proof.

Theorem 7. *Let* COM *be a secure commitment scheme, and let* NIZK *be a non-interactive zero knowledge proof. If Schnorr signatures are strongly existentially unforgeable, then the construction in Fig. D.2 UC-realizes the ideal functionality $\mathcal{F}$ in the $(\mathcal{F}^{\mathsf{schnorr}}_{\mathsf{kgen}}, \mathcal{F}_{\mathsf{syn}}, \mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{anon}})$-hybrid model.*

D.4 Comparison of Privacy Notions and Guarantees

In this section, we discuss our notion of relationship anonymity as the privacy notion of interest for PCNs and compare it with other possible privacy notions described in the literature related to PCN.

Our privacy model faithfully captures the reality of the currently deployed PCNs. In particular, Malavolta et al. [MMSK+17] showed that it captures the well-established notion of relationship anonymity. In a nutshell, relationship anonymity [PH10] requires that, given two simultaneous successful payment operations between $\text{sender}_{\{0,1\}}$ and $\text{receiver}_{\{0,1\}}$ that share the same path with at least one honest intermediate user, corrupted intermediate users cannot determine the correct pair (sender_b, receiver_b) for a given payment with probability better than $1/2$ (i.e., guessing). Note that this holds only for payments of the same value since such information is trivially leaked to intermediate users, i.e., each user can monitor how adjacent links evolve and infer the amount that was transferred.

An alternative privacy notion is described in BOLT [GM17]. There, authors propose *payment anonymity*. Intuitively, payment anonymity requires that the merchant, even in collaboration with a set of malicious customers, learns nothing about a customer's spending pattern beyond the information available outside the payment protocol.

While this privacy notion additionally hides the value that is transacted, it is restricted to single-hop payments and does not consider the crucial aspect of conditional payment required when more than one intermediate user takes part in the payment. As discussed in Section 5.3, many well-established networks use paths with multiple intermediaries, and it is reasonable to expect long paths also in the LN. To obtain the best of both worlds, one could envision a protocol where private one-hop payments are performed "at the edges" (i.e., between the sender and the first hop as well as between last hop and the receiver) while the rest of intermediate users carry out a multi-hop payment à la LN.

However, this approach raises several questions. First, it is unclear whether the hypothetical resulting privacy guarantees are stronger or weaker than those presented in this work. It is possible that the naïve combination of the two systems would completely break down the guarantees of both schemes. Techniques presented in both works might be required to develop a new system. Second, BOLT requires a blockchain supporting a rich scripting language, and it is therefore not compatible with prominent cryptocurrencies (such as Bitcoin). Thus, making this system Bitcoin-compatible would require fundamentally new techniques.

In summary, although it seems to be an interesting research direction; further work is required to study this approach and its privacy properties.

D.5 Security Analysis

Throughout the analysis, we denote by $\mathsf{poly}(\lambda)$ any function that is bounded by a polynomial in λ. We denote any function that is negligible in the security parameter by $\mathsf{negl}(\lambda)$. We say that

an algorithm is PPT if it is modeled as a probabilistic Turing machine whose running time is bounded by some function $\mathsf{poly}(\lambda)$.

In the following we recall the (non standard) ideal functionalities that our protocols build on and we elaborate on the security analysis of our constructions. We stress that the copies of this functionality that are invoked as subroutines are fresh independent instances and therefore, the composition theorem [Can01] directly applies to our settings.

Key Generation Functionalities. Our ideal functionality for the key generation of Schnorr signatures $\mathcal{F}_{\mathsf{keygen}}^{\mathsf{schnorr}}$ provides the users with the interface described below. This essentially models a distributed key generation for discrete logarithm-based schemes, which is a very well-studied problem (see, e.g., [GJKR07]).

KeyGen($\mathbb{G}$, G, q)

Upon invocation by both U_0 and U_1 on input ($\mathbb{G}$, G, q):
sample $x \leftarrow\!\!\$\, \mathbb{Z}_q$ and compute $Q = x \cdot G$
set $\mathsf{sk}_{U_0,U_1} = x$
sample x_0 and x_1 randomly
sample a hash function $H : \{0,1\}^* \to \{0,1\}^{|q|}$
send (x_0, Q, H) to U_0 and (x_1, Q, H) to U_1
ignore future calls by (U_0, U_1)

Our ideal functionality for the key generation of ECDSA $\mathcal{F}_{\mathsf{keygen}}^{\mathsf{ECDSA}}$ is taken almost in verbatim from [Lin17], and it is given in the following.

KeyGen($\mathbb{G}$, G, q)

Upon invocation by both U_0 and U_1 on input ($\mathbb{G}$, G, q):
sample $x \leftarrow\!\!\$\, \mathbb{Z}_q$ and compute $Q = x \cdot G$
sample x_0 and x_1 randomly
sample a hash function $H : \{0,1\}^* \to \{0,1\}^{|q|}$
sample a key pair $(\mathsf{sk}_{U_0,U_1}, \mathsf{pk}_{U_0,U_1}) \leftarrow \mathsf{KGen}_{\mathsf{HE}}(1^\lambda)$
compute $c \leftarrow \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, \tilde{r})$ for a random $\tilde{r}$
send (x_0, Q, H, sk) to U_0 and (x_1, Q, H, c) to U_1
ignore future calls by (U_0, U_1)

Generic Construction. Here we elaborate on the proof of Theorem 2.

Proof. We define the following sequence of hybrids, where we gradually modify the initial experiment.

$\mathcal{H}_0$: Is identical to the protocol as described in Section 5.5.2.

$\mathcal{H}_1$: Consider the following ensemble of variables in the interaction with $\mathcal{A}$: A honest user U_i, a key pair $(\mathsf{sk}_i, \mathsf{pk})$, a state s^I, a tuple $(\ell_i, \ell_{i+1}, s^L, s^R)$ such that

$$\{\cdot, (\ell_i, s^L)\} \leftarrow \langle \cdot, \mathsf{Lock}_{U_i}(s^I, \mathsf{sk}_i, \mathsf{pk})\rangle$$

and

$$\{(\ell_{i+1}, s^R), \cdot\} \leftarrow \langle \mathsf{Lock}_{U_i}(s^I, \mathsf{sk}_i, \mathsf{pk}), \cdot\rangle.$$

If, for any set of these variables, the adversary returns some k such that $\mathsf{Vf}(\ell_{i+1}, k) = 1$ and $\mathsf{Vf}(\ell_i, \mathsf{Rel}(k, (s^I, s^L, s^R))) \neq 1$, then the experiment aborts.

$\mathcal{H}_2$: Consider the following ensemble of variables in the interaction with $\mathcal{A}$: A pair of honest users (U_0, U_i) a set of (possibly corrupted) users $(U_1, \ldots, U_n)$, a key pair $(\mathsf{sk}_i, \mathsf{pk})$, a set of initial states

$$(s_0^I \ldots, s_n^I) \leftarrow \left\langle \begin{array}{c} \mathsf{Setup}_{U_0}(1^\lambda, U_1, \ldots, U_n), \\ \ldots, \\ \mathsf{Setup}_{U_n}(1^\lambda) \end{array} \right\rangle,$$

and a pair of locks (ℓ_{i-1}, ℓ_i) such that

$$\{\cdot, (\ell_{i-1}, \cdot)\} \leftarrow \langle \cdot, \mathsf{Lock}_{U_i}(s_i^I, \mathsf{sk}_i, \mathsf{pk})\rangle$$

and

$$\{(\ell_i, \cdot), \cdot\} \leftarrow \langle \mathsf{Lock}_{U_i}(s_i^I, \mathsf{sk}_i, \mathsf{pk}), \cdot\rangle.$$

If, for any set of these variables, the adversary returns some k_{i-1} such that $\mathsf{Vf}(\ell_{i-1}, k_{i-1}) = 1$ before the user U_i outputs a key k_i such that $\mathsf{Vf}(\ell_i, k_i) = 1$, then the experiment aborts.

$\mathcal{H}_3$: Let $S = (U_0, \ldots, U_m)$ be an ordered set of (possibly corrupted) users. We say that that an ordered subset $A = (U_1, \ldots, U_j)$ is *adversarial* if U_i is honest and $(U_{i+1}, \ldots, U_j)$ are corrupted. Note that every set of users can be expressed as a concatenation of adversarial subsets, that is $S = (A_1 || \ldots || A_{m'})$, for some $m' \leq m$. Whenever a honest user is requested to set up a lock for a certain set $S = (A_1 || \ldots || A_{m'})$, it initializes an independent lock for each subset (A_i, A_{i+1}^0), where A_{i+1}^0 is the first element of the $(i+1)$-th set, if present. Whenever some A_{i+1}^0 is requested to release the key for the corresponding lock (recall that all A_{i+1}^0 are honest nodes) it releases the key for the fresh lock (A_i, A_{i+1}^0) instead.

$\mathcal{S}$: The interaction of the simulator is identical to $\mathcal{H}_3$ except that the actions of $\mathcal{S}$ are dictated by the interaction with $\mathcal{F}$. The simulator reads the communication of $\mathcal{A}$ with the honest users via $\mathcal{F}_{\mathsf{anon}}$ and is queried by $\mathcal{F}$ on the following set of inputs.

1. $(\cdot, \cdot, \cdot, \cdot, \mathsf{Init})$: The simulator reconstructs the adversarial set (defined above) from the ids and sets up a fresh lock chain.
2. $(\cdot, \mathsf{Lock})$: The simulator initiates the locking procedure with the adversary and replies with $\perp$ if the execution is not successful.
3. $(\cdot, \mathsf{Rel})$ The simulator releases the key of the corresponding lock and publishes it.

If $\mathcal{A}$ interacts with an honest user (e.g., by releasing a lock), the simulator queries the corresponding interface of $\mathcal{F}$.

Note that the simulator is efficient and interacts as the adversary with the ideal world. Furthermore, the simulation is always consistent with the ideal world, i.e., if the adversary's action is not supported by the interfaces of $\mathcal{F}$, the simulation aborts. What is left to be shown is that the neighboring hybrids are indistinguishable to the eyes of the environment $\mathcal{E}$.

Lemma 1. *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_0,\mathcal{A},\mathcal{E}} \equiv \mathsf{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}}.$$

Proof. Follows from the homomorphic property of the function g: Recall that a key-lock pair (k, ℓ) is valid if and only if $g(k) = \ell$. Let (k_i, ℓ_i) be the output of $\mathcal{A}$, by construction we have that $\ell_i = \ell_{i-1} + g(y_i)$, for some (ℓ_{i-1}, y_i), which is part of the state of the honest node. Since the release algorithm computes $k_i - y_i$ we have that

$$\begin{aligned} g(k_i - y_i) &= g(k_i) - g(y_i) \\ &= \ell_i - g(y_i) \\ &= \ell_{i-1} + g(y_i) - g(y_i) \\ &= \ell_{i-1} \end{aligned}$$

with probability 1, by the homomorphic property of g. □

Lemma 2. *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}}.$$

Proof. Let $q \in \mathsf{poly}(\lambda)$ be a bound on the number of interactions. Recall that $\mathcal{H}_1$ and $\mathcal{H}_2$ differ only for the case where the adversary outputs a key for a honestly generated lock before the trapdoor is released. Assuming towards contradiction that the probability that this event happens is non-negligible, we can construct the following reduction against the one-wayness of g: On input some $Y^* \in \mathcal{R}$, the reduction guesses a session $j \in [1, q]$ and some index $i \in [1, n]$. The setup algorithm of the j-th session is modified as follows: Y_i is set to be Y^*. Then, for all $\iota \in [i-1, 0]$, the setup samples some $y_\iota \in \mathcal{D}$ and returns $(Y_\iota = Y_{\iota+1} - g(y_\iota), Y_{\iota+1}, y_\iota)$. The setup samples a random $y_i \in \mathcal{D}$ and sets $Y_{i+1} = g(y_i)$. Then, for $\iota \in [i+1, n-1]$, the setup samples $y_\iota \in \mathcal{D}$ returns $(Y_\iota, Y_\iota + g(y_\iota), y_\iota)$. The nodes $(U_1, \ldots, U_{n-1})$ are given the corresponding output (except for U_i) and U_n is given $(Y_{n-1}, \sum_{j=i}^{n-1} y_j)$. If the node U_i is requested to release the lock, the reduction aborts. At some point of the execution the adversary $\mathcal{A}$ outputs some $y*$, and the reduction returns $y^* + y_{i-1}$.

The reduction is clearly efficient and, whenever j and i are guessed correctly, the reduction does not abort. Since the group defined by g is abelian, the distribution induced by the modified setup algorithm is identical to the original (except for the initial state of U_1). Also note that, whenever j and i are guessed correctly, the user U_i is honest and therefore the adversary does not not see

the corresponding internal state. It follows that the reduction is identical to $\mathcal{H}_1$, to the eyes of the adversary. Finally, whenever the adversary outputs some valid k_{i-1} for ℓ_{i-1}, then it holds that $g(k_{i-1}) = \ell_{i-1}$. Substituting we have that

$$\begin{aligned} g(k_{i-1}) &= \ell_{i-1} \\ g(y^*) &= Y_{i-1} \\ g(y^*) &= Y^* - g(y_{i-1}) \\ g(y^*) + g(y_{i-1}) &= Y^* \\ g(y^* + y_{i-1}) &= Y^*. \end{aligned}$$

It follows that the reduction is successful with probability at least $\frac{1}{q \cdot n \cdot \mathsf{poly}(\lambda)}$. This proves our statement. □

Lemma 3. *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}} \equiv \mathsf{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}}.$$

Proof. Recall that adversarial sets are always interleaved by a honest node. Therefore in $\mathcal{H}_2$ for each adversarial set starting at index i there exists a y such that $Y_i = Y_{i-1} + g(y)$ and $\mathcal{A}$ is not given y. Since y is randomly sampled from $\mathcal{D}$ we have that $Y_{i-1} + g(y) \equiv Y'$, for some Y' sampled uniformly from $\mathcal{R}$, which corresponds to the view of $\mathcal{A}$ in $\mathcal{H}_3$. □

Lemma 4. *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}} \equiv \mathsf{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{E}}.$$

Proof. The changes between the two experiments are only conceptual and the equivalence of the views follows. □

This concludes our analysis. □

Schnorr-based Construction. Here we prove Theorem 7.

Proof. We define the following sequence of hybrids, where we gradually modify the initial experiment.

$\mathcal{H}_0$: Is identical to the protocol as described in Appendix D.3.

$\mathcal{H}_1$: All the calls to the commitment scheme are replaced with interactions with the ideal functionality $\mathcal{F}_{\mathsf{com}}$, defined in the following.

Commit(*sid*, m)

Upon invocation by U_i (for $i \in \{0,1\}$):
record (sid, i, m) and send (com, sid) to U_{1-i}
if some $(sid, \cdot, \cdot)$ is already stored ignore the message

Decommit(*sid*)

Upon invocation by U_i (for $i \in \{0,1\}$):
if (sid, i, m) is recorded then send (decom, sid, m) to U_{1-i}

Instead of calling the Commit algorithm on some message m, the parties sent a message of the form **Commit(sid, m)** to the ideal functionality, and the decommitment algorithm is replaced with a call to **Decommit(sid)**. The verifying party simply records messages from $\mathcal{F}_{\mathsf{com}}$.

$\mathcal{H}_2$: All the calls to the NIZK scheme are replaced with interactions with the ideal functionality $\mathcal{F}_{\mathsf{NIZK}}$:

Prove(*sid*, x, w)

Upon invocation by U_i (for $i \in \{0,1\}$):
if $R(x,w) = 1$ then send (proof, sid, x) to U_{1-i}

Instead of running the proving algorithm in input (x, w), the proving party queries the functionality on **Prove(sid, x, w)**. The verifier records the messages from $\mathcal{F}_{\mathsf{NIZK}}$.

$\mathcal{H}_3, \mathcal{H}_4, \mathcal{H}_5, \mathcal{S}$: The subsequent hybrids are defined as $\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3, \mathcal{S}$, respectively, in Theorem 2.

As argued before, the simulator is efficient, and the interaction is consistent with the inputs of the ideal functionality. In the following, we prove the indistinguishability of the neighboring experiments.

Lemma 5. *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_0,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}}.$$

Proof. Follows directly from the security of the commitments scheme COM. □

Lemma 6. *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}}.$$

Proof. Follows directly from the security of the non-interactive zero-knowledge scheme NIZK. □

Lemma 7. *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}}.$$

Proof. In order to show this claim, we introduce an intermediate experiment.

$\mathcal{H}_2^*$: The locking algorithms are substituted with the following ideal functionality. Such an interface is called by both parties on input m and $y = \sum_{j=0}^{i} y_j$, where i is the position of the

lock in the chains and the y_j are defined as in the original protocol. Note that the key sk_{U_0,U_1} refers to the previously established key in the call to the $\mathcal{F}_{\mathsf{kgen}}^{\mathsf{schnorr}}$.

Sign(m, y)

Upon invocation by both U_0 and U_1 on input (m, y):
compute $(R, s) = \mathsf{Sig}_{\mathsf{schnorr}}(\mathsf{sk}_{U_0,U_1}, m)$
return $(R, s - y)$

We defer the indistinguishability proof to Lemma 8. Let cheat by the event that triggers an abort of the experiment in $\mathcal{H}_3$, that is, the adversary returns some k such that $\mathsf{Vf}(\ell_{i+1}, k) = 1$ and that $\mathsf{Vf}(\ell_i, \mathsf{Rel}(k, (s^I, s^L, s^R))) \neq 1$. Assume towards contradiction that $\Pr[\mathsf{cheat} \mid \mathcal{H}_2^*] \geq \frac{1}{\mathsf{poly}(\lambda)}$, then we can construct the following reduction against the strong-existential unforgeability of Schnorr signatures: The reduction receives as input a public key pk and samples an index $j \in [1, q]$, where $q \in \mathsf{poly}(\lambda)$ is a bound on the total amount of interactions. Let Q be the key generated in the j-th interaction, the reduction sets $Q = \mathsf{pk}$. All the calls to the signing algorithm are redirected to the signing oracle. If the event cheat happens, the reduction returns corresponding $(k^*, \ell^*) = (\sigma^*, (m^*, \mathsf{pk}^*))$, otherwise it aborts.

The reduction is clearly efficient. Assume for the moment that j is the index of the interaction where cheat happens, and let $i+1$ be the index that identifies the lock ℓ^* in the corresponding chain. Note that in case the guess of the reduction is correct we have that $\mathsf{pk}^* = \mathsf{pk}$. Since cheat happens we have that $\mathsf{Vf}_{\mathsf{schnorr}}(\mathsf{pk}^*, m^*, \sigma^*) = 1$ and the release fails, i.e., $\mathsf{Vf}(\ell_i, \mathsf{Rel}(k, (s_i^I, s_i^L, s_i^R)) \neq 1$ (where ℓ_i is the lock in the previous position as ℓ^* in the same chain). Recall that the release algorithm parses s_i^L as $(W_{i,0}, w_{i,1})$ and σ^* as (R^*, s^*) and returns $(W_{i,0}, w_{i,1} + s^* - (s_i^R + y_i))$. Substituting with the corresponding values

$$\begin{aligned}
&\left(W_{i,0}, w_{i,1} + s^* - (s_i^R + y_i)\right) \\
&= \left(R_i, s_i - \sum_{j=0}^{i-1} y_j + s^* - s_j - \sum_{j=0}^{i} y_j + y_i\right) \\
&= \left(R_i, s_i + s^* - s_j\right),
\end{aligned}$$

where s_j is the answer of the oracle on the j-th session on input m_j. This implies that $s^* \neq s_j$, otherwise (R_i, s_i) would be a valid signature since it is an output of the signing oracle. Since each message uniquely identifies a session (the same message is never queried twice to the interface **Sign**(m,y)) this implies that $(\sigma^*, (m^*, \mathsf{pk}^*))$ is a valid forgery. By assumption this happens with probability at least $\frac{1}{q \cdot \mathsf{poly}(\lambda)}$, which is a contradiction and proves that $\Pr[\mathsf{cheat} \mid \mathcal{H}_2^*] \leq \mathsf{negl}(\lambda)$. Since the experiments $\mathcal{H}_2$ and $\mathcal{H}_3$ differ only when cheat happens (and $\mathcal{H}_3$ aborts), we are only left with showing the indistinguishability of $\mathcal{H}_2$ and $\mathcal{H}_2^*$.

Lemma 8. *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_2^*,\mathcal{A},\mathcal{E}}.$$

Proof. The proof consists of the description of the simulator for the interactive lock algorithm. We describe two simulators depending on whether the honest adversary is playing the role of the "left" or "right" party. For each proof, both the simulators implicitly check that the given witness is valid and abort if this is not the case.

1. Left corrupted: Prior to the interaction the simulator is sent $(Y, y, (\mathsf{prove}, \{\exists y^* \text{ s.t } y^* \cdot G = Y\}, y^*))$, which is the state corresponding to the execution of the lock. After agreeing on a message m, the simulator sends $(\mathsf{com}, \mathit{sid})$ to $\mathcal{A}$, for a random sid. The simulator also queries the interface **Sign** on input m, y^* and receives a signature $\sigma = (R, s)$. At some point of the execution $\mathcal{A}$ sends $(R_0, (\mathsf{prove}, \{\exists r_0 \text{ s.t } r_0 \cdot G = R_0\}, r_0))$. The simulator replies with
$$\left(\begin{array}{l} \mathsf{decom}, \mathit{sid}, \left(\begin{array}{l} R^* = R - (R_0 + Y), \\ \mathsf{proof}, \mathit{sid}, \\ \{\exists r^* \text{ s.t } r^* \cdot G = R^*\} \end{array}\right), \\ R^*, (s - r_0 - e \cdot x_0) \end{array}\right)$$
where $e = H(\mathsf{pk} \| R^* \| m)$ and x_0 is the value returned by the key generation to $\mathcal{A}$. The rest of the execution is unchanged.

2. Right corrupted: Prior to the interaction the simulator is sent $(Y, y, (\mathsf{prove}, \{\exists y^* \text{ s.t } y^* \cdot G = Y\}, y^*))$, which is the state corresponding to the execution of the lock. After agreeing on a message m, the simulator is given
$$\left(\mathsf{com}, \mathit{sid}, \left(\begin{array}{ll} R_1, & \mathsf{prove}, \mathit{sid}, \\ & \{\exists r_1 \text{ s.t } r_1 \cdot G = R_1\}, r_1 \end{array}\right)\right)$$
by $\mathcal{A}$. The simulator then queries the interface **Sign** on input m, y^* and receives a signature $\sigma = (R, s)$. The simulator sends $(R^* = R - (R_1 + Y), (\mathsf{proof}, \mathit{sid}, \{\exists r^* \text{ s.t } r^* \cdot G = R^*\}))$ to $\mathcal{A}$ and receives $((\mathsf{decom}, \mathit{sid}), s^*)$ in response. The simulator checks whether $s^* = r_1 + e \cdot x_1$, where $e = H(\mathsf{pk} \| R^* \| m)$, and returns s if this is the case.

Both simulators are obviously efficient and the distributions induced by the simulated views are identical to the ones of the original protocol. □

This concludes the proof of Lemma 7. □

Lemma 9. *For all PPT distinguishers $\mathcal{E}$ it holds that*
$$\mathsf{EXEC}_{\mathcal{H}_3, \mathcal{A}, \mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_4, \mathcal{A}, \mathcal{E}}.$$

Proof. Let $q \in \mathsf{poly}(\lambda)$ be a bound on the number of interactions. Let cheat denote the events that triggers an abort in $\mathcal{H}_4$ but not in $\mathcal{H}_3$. In the following we are going to show that $\Pr[\mathsf{cheat} \mid \mathcal{H}_3] \leq \mathsf{negl}(\lambda)$, thus proving the indistinguishability of $\mathcal{H}_3$ and $\mathcal{H}_4$. Assume that the converse is true, then we can construct the following reduction against the discrete logarithm problem (which is implied by the sEUF of Schnorr): On input some $Y^* \in \mathbb{G}$, the reduction guesses a session

$j \in [1, q]$ and some index $i \in [1, n]$. The setup algorithm of the j-th session is modified as follows: Y_i is set to be Y^*. Then, for all $\iota \in [i-1, 0]$, the setup samples some $y_\iota \in \mathbb{Z}_q$ and returns $(Y_\iota = Y_{\iota+1} - y_\iota \cdot G, Y_{\iota+1}, y_\iota)$. The setup samples a random $y_i \in \mathbb{Z}_q$ and sets $Y_{i+1} = y_i \cdot G$. Then, for $\iota \in [i+1, n-1]$, the setup samples $y_\iota \in \mathbb{Z}_q$ returns $(Y_\iota, Y_\iota + y_\iota \cdot G, y_\iota)$. The nodes $(U_1, \ldots, U_{n-1})$ are given the corresponding output (except for U_i) and U_n is given $(Y_{n-1}, \sum_{j=i}^{n-1} y_j)$. If the node U_i is requested to release the lock, the reduction aborts. At some point of the execution the adversary $\mathcal{A}$ outputs some $k^* = (R^*, s^*)$. The reduction parses s^R as the updated state of U_i and returns $s^* + y_{i-1} - s^R$.

The reduction is clearly efficient and, whenever j and i are guessed correctly, the reduction does not abort. Since the group $\mathbb{G}$ is abelian and the U_i is honest, the distribution induced by the modified setup algorithm is identical to the original to the eyes of the adversary. Recall that cheat happens only in the case where k^* is a valid opening for ℓ_i and the release algorithm is successful on input k^* (if the last condition is not satisfied both $\mathcal{H}_3$ and $\mathcal{H}_4$ abort). Substituting, we have that s^R is of the form $r_0 + r_1 + e \cdot (x0 + x_1) - y = s' - y$, for some $y \in \mathbb{Z}_q$. Since the release is successful, then it must be the case that $(R' = (r_0 + r_1) \cdot G + Y_{i-1}, s')$ is a valid Schnorr signature on the message m_{i-1} (agreed by the two parties in the locking algorithm for ℓ_{i-1}), which implies that $y \cdot G = Y_{i-1}$. As argued in the proof of Lemma 7, if $s^* \neq s'$, then we have an attacker against the strong unforgeability of the signature scheme. It follows that $s^* = s'$ with all but negligible probability. Substituting we have

$$\begin{aligned}(s^* + y_{i-1} - s^R) \cdot G &= (s^* + y_{i-1} - s' + y) \cdot G \\ &= (y_{i-1} + y) \cdot G \\ &= y_{i-1} \cdot G + y \cdot G \\ &= y_{i-1} \cdot G + Y_{i-1} \\ &= y_{i-1} \cdot G + (Y^* - y_{i-1} \cdot G) \\ &= Y^*\end{aligned}$$

as expected. Since, by assumption, this happens with probability at least $\frac{1}{q \cdot n \cdot \mathsf{poly}(\lambda)}$ we have a successful attacker against the discrete logarithm problem. This proves our statement.

□

Lemma 10. *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_4, \mathcal{A}, \mathcal{E}} \equiv \mathsf{EXEC}_{\mathcal{H}_5, \mathcal{A}, \mathcal{E}}.$$

Proof. Recall that adversarial sets are always interleaved by a honest node. Therefore in $\mathcal{H}_4$ for each adversarial set starting at index i there exists a y such that $Y_i = Y_{i-1} + y \cdot G$ and $\mathcal{A}$ is not given y. Since y is randomly sampled from $\mathbb{Z}_q$ we have that $Y + i - 1 + y \cdot G \equiv Y'$, for some Y' sampled uniformly from $\mathbb{G}$, which corresponds to the view of $\mathcal{A}$ in $\mathcal{H}_5$. □

Lemma 11. *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_5, \mathcal{A}, \mathcal{E}} \equiv \mathsf{EXEC}_{\mathcal{F}, \mathcal{S}, \mathcal{E}}.$$

Proof. The change is only syntactical, and the indistinguishability follows. □

This concludes our analysis. □

ECDSA-based Construction. In the following we prove Theorem 3.

Proof. The sequence of hybrids that we define is identical to the one described in the proof of Theorem 7. In the following, we prove the indistinguishability of neighboring experiments only for the cases where the argument needs to be modified. If the argument is identical, the proof is omitted.

Lemma 12. *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}}.$$

Proof. In order to show this claim, we introduce an intermediate experiment.

$\mathcal{H}_2^*$: The locking algorithms is substituted with the interaction with the following ideal functionality. Recall that the key sk_{U_0,U_1} here refers to the key established during the call of the same pair of users to the key generation functionality. Also, note that the locking algorithm is called by both parties on input m and $y = \sum_{j=0}^{i} y_j$, where i is the position of the lock in the chains and the y_j are defined as in the original protocol.

Sign(m, y)

Upon invocation by both U_0 and U_1 on input (m, y):
compute $(r, s) = \mathsf{Sig}_{\mathsf{ECDSA}}(\mathsf{sk}_{U_0,U_1}, m)$
return $(r, \mathsf{min}(s \cdot y, -s \cdot y))$

The indistinguishability proof of $\mathcal{H}_2$ and $\mathcal{H}_2^*$ is formally shown in Lemma 13. Let cheat by the event that triggers an abort of the experiment in $\mathcal{H}_3$, that is, the adversary returns some k such that $\mathsf{Vf}(\ell_{i+1}, k) = 1$ and $\mathsf{Vf}(\ell_i, \mathsf{Rel}(k, (s^I, s^L, s^R))) \neq 1$. Assume towards contradiction that $\mathsf{Pr}\left[\mathsf{cheat} \mid \mathcal{H}_2^*\right] \geq \frac{1}{\mathsf{poly}(\lambda)}$, then we can construct the following reduction against the strong-existential unforgeability of ECDSA signatures: The reduction receives as input a public key pk and samples an index $j \in [1, q]$, where $q \in \mathsf{poly}(\lambda)$ is a bound on the total amount of interactions. Let Q be the key generated in the j-th interaction, the reduction sets $Q = \mathsf{pk}$. All the calls to the signing algorithm are redirected to the signing oracle. If the event cheat happens, the reduction returns corresponding $(k^*, \ell^*) = (\sigma^*, (m^*, \mathsf{pk}^*))$, otherwise it aborts.

The reduction runs in polynomial time. Assume for the moment that j is the index of the interaction where cheat happens, and let $i + 1$ be the index that identifies the lock ℓ^* in the corresponding chain. Note that in case the guess of the reduction is correct we have that $\mathsf{pk}^* = \mathsf{pk}$. Since cheat happens we have that $\mathsf{Vf}_{\mathsf{ECDSA}}(\mathsf{pk}^*, m^*, \sigma^*) = 1$ and the release fails, i.e., $\mathsf{Vf}(\ell_i, \mathsf{Rel}(k^*, k^*, (s_i^I, s_i^L, s_i^R))) \neq 1$ (where ℓ_i is the lock in the previous position as ℓ^* in the same chain). Recall that the release algorithm parses s_i^L as $(w_{i,0}, w_{i,1})$, σ^* as (r^*, s^*), and

s_i^R as (s', m, pk) and computes $t = w_1 \cdot (\frac{s'}{s^*} - y)^{-1}$ and $t' = w_1 \cdot (-\frac{s'}{s^*} - y)^{-1}$. Then it returns either $(w_{i,0}, \mathsf{min}(t, -t))$ or $(w_{i,0}, \mathsf{min}(t', -t'))$ depending on which verifies as a valid signature on m under pk. Substituting with the corresponding values (for the case t is the lower term)

$$\begin{aligned}(w_{i,0}, t) &= \left(r_i, w_{i,1} \cdot \left(\frac{s'}{s^*} - y\right)^{-1}\right) \\ &= \left(r_i, s_i \cdot \sum_{j=0}^{i-1} y_j \cdot \left(\frac{s_j \cdot \sum_{j=0}^{i} y_j}{s^*} - y_i\right)^{-1}\right)\end{aligned}$$

where s_j is the answer of the oracle on the j-th session on input the corresponding message m_j. If we set $s^* = s_j$ then we have

$$\begin{aligned}(w_{i,0}, t) &= \left(r_i, s_i \cdot \sum_{j=0}^{i-1} y_j \cdot \left(\sum_{j=0}^{i} y_j - y_i\right)^{-1}\right) \\ &= (r_i, s_i)\end{aligned}$$

which is a valid signature on m_i (since it is the output of the signing oracle) and the release would be successful. So this cannot happen and we can assume that $s^* \neq s_j$. A similar argument (substituting t with t') can be used to show that it must be the case that $s^* \neq -s_j$. Since each message uniquely identifies a session (the same message is never queried twice to the interface **Sign**(m,y)) this implies that $(\sigma^*, (m^*, \mathsf{pk}^*))$ is a valid forgery. By assumption this happens with probability at least $\frac{1}{q \cdot \mathsf{poly}(\lambda)}$, which is a contradiction and proves that $\mathsf{Pr}\left[\mathsf{cheat} \mid \mathcal{H}_2^*\right] \leq \mathsf{negl}(\lambda)$. Since the experiments $\mathcal{H}_2$ and $\mathcal{H}_3$ differ only when cheat happens (and $\mathcal{H}_3$ aborts), we are only left with showing the indistinguishability of $\mathcal{H}_2$ and $\mathcal{H}_2^*$.

Lemma 13. *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_2, \mathcal{A}, \mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_2^*, \mathcal{A}, \mathcal{E}}.$$

Proof. The proof consists of the description of the simulator for the interactive lock algorithm. In the following, we describe the two simulators for the locking protocol depending on whether the honest adversary is playing the role of the "left" or "right" party. For each zero-knowledge proof, both the simulators implicitly check that the given witness is valid and abort if this is not the case.

1. Left corrupted: Prior to the interaction the simulator is sent $(Y, y, (\mathsf{prove}, \{\exists y^* \text{ s.t } y^* \cdot G = Y\}, y^*))$, which is the state corresponding to the execution of the lock. After agreeing on a message m, the simulator sends (com, sid) to $\mathcal{A}$, for a random sid. The simulator also queries the interface **Sign** on input m, y^* and receives a signature $\sigma = (r, s)$. The simulator sets $R = \frac{H(m)}{s} \cdot G + \frac{r}{s} \cdot \mathsf{pk}$. At some point of the execution $\mathcal{A}$ sends $(R_0, R_0', (\mathsf{prove}, \{\exists r_0 \text{ s.t } r_0 \cdot G = R_0 \text{ and } r_0 \cdot Y = R_0'\}, r_0))$. Then the simulator samples

a $\rho \leftarrow \mathbb{Z}_{q^2}$ and computes $c' \leftarrow \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, s \cdot r_0 + \rho q)$. Then it provides the attacker with

$$\mathsf{decom}, \mathit{sid}, \left(\begin{array}{l} R^* = (r_0)^{-1} \cdot R, \\ R_1 = y^{-1} \cdot R^*, \\ \mathsf{proof}, \mathit{sid}, \\ \left\{ \begin{array}{l} \exists r^* \text{ s.t } r^* \cdot G = R_1 \\ \quad \text{and } r^* \cdot Y = R^* \end{array} \right\} \end{array} \right), \\ R_1, R^*, c'.$$

The rest of the execution is unchanged.

The executions are identical except for the way c' is computed. In order to show the statistical proximity, we invoke the following helping lemma.

Lemma 14. *[Lin17] For all $(r, s, p) \in \mathbb{Z}_q$ and for a random $\rho \in \mathbb{Z}_{q^2}$, the distributions $\mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, r \cdot s \bmod q + pq + \rho q)$ and $\mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, r \cdot s \bmod q + \rho q)$ are statistically close.*

In the real world c' is computed as $\mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, r \cdot s \bmod q + pq + \rho q)$, for some p which is bounded by q since the only operation performed without modular reduction are one multiplication and one addition, which cannot increase the result by more than q^2. Since the distribution $\mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, r \cdot s \bmod q + \rho q)$ is identical to the simulation, the indistinguishability follows.

2. Right corrupted: Prior to the interaction the simulator is sent $(Y, y, (\mathsf{prove}, \{\exists y^* \text{ s.t } y^* \cdot G = Y\}, y^*))$, which is the state corresponding to the execution of the lock. After agreeing on a message m, the simulator is given

$$\mathsf{com}, \mathit{sid}, \\ \left(R_1, R_1', \begin{array}{l} \mathsf{prove}, \mathit{sid}, \\ \left\{ \begin{array}{l} \exists r_1 \text{ s.t } r_1 \cdot G = R_1 \text{ and} \\ \quad r_1 \cdot Y = R_1' \end{array} \right\}, \\ r_1 \end{array} \right)$$

by $\mathcal{A}$. The simulator then queries the interface **Sign** on input m, y^* and receives a signature $\sigma = (r, s)$. Then it sets $R = \frac{H(m)}{s} \cdot G + \frac{r}{s} \cdot \mathsf{pk}$ and $R^* = R - (R_1 + Y)$ and sends $(R_0 = y^{-1} \cdot R^*, R^*, (\mathsf{proof}, \mathit{sid}, \{\exists r^* \text{ s.t } r^* \cdot G = R_0 \text{ and } r^* \cdot Y = R^*\}))$ to $\mathcal{A}$. The attacker sends $((\mathsf{decom}, \mathit{sid}), c')$ in response. The simulator checks

$$\mathsf{Dec}_{\mathsf{HE}}(\mathsf{sk}, c') = \tilde{r} \cdot r \cdot (r_1)^{-1} + H(m) \cdot r_1^{-1} \bmod q,$$

where $\tilde{r}$ was sampled in the key generation algorithm. If the check holds true, the simulator sends s to $\mathcal{A}$.

The distribution induced by the simulator is identical to the real experiment except for the way c is computed. Towards showing indistinguishability, consider the following modified simulator,

that is given the oracle $\mathcal{O}(c', a, b)$ as defined in the following security experiment of the Paillier encryption scheme.

$\mathsf{Exp} - \mathsf{ecCPA}^{\mathcal{A}}_{\mathsf{HE}}(\lambda)$:

$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}_{\mathsf{HE}}(1^\lambda)$
$(w_0, w_1) \leftarrow\!\!\$\, \mathbb{Z}_q$
$Q = w_0 \cdot G$
$b \leftarrow\!\!\$\, \{0, 1\}$
$c \leftarrow \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, w_b)$
$b' \leftarrow \mathcal{A}(\mathsf{pk}, c, Q)^{\mathcal{O}(\cdot,\cdot,\cdot)}$
where $\mathcal{O}(c', a, b)$ returns 1 iff $\mathsf{Dec}_{\mathsf{HE}}(\mathsf{sk}, c') = a + b \cdot w_b$
return 1 iff $b = b'$

Instead of performing the last check, the simulator queries the oracle on input $(c', a = H(m) \cdot r_1^{-1}, b = r \cdot (r_1)^{-1})$. It is clear that the modified simulator accepts if and only if the simulator described above accepts. Assume towards contradiction that the modified simulator can be efficiently distinguished from the real-world experiment. Then we can reduce to the security of Paillier as follows: On input (pk, c, Q), the reduction simulates the inputs of $\mathcal{A}$ as described in the modified simulator using the input pk, Q, and c as the corresponding variables. It is easy to see that the reduction is efficient. Note that if $b = 0$ then we have that $c = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, w_0)$ and $Q = w_0 \cdot G$, which is identical to the real-world execution. On the other hand if $b = 1$ then it holds that $c = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, w_1)$ and $Q = w_0 \cdot G$, where w_1 is uniformly distributed in $\mathbb{Z}_q$, which is identical to the (modified) simulated experiment. This implies that the modified simulation is computationally indistinguishable from the real-world experiment. Since the modified simulation and the simulation (as described above) are identical to the eyes of the adversary, the validity of the lemma follows. □

This concludes the proof of Lemma 12. □

Lemma 15. *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}}.$$

Proof. Let $q \in \mathsf{poly}(\lambda)$ be a bound on the number of interactions. Let cheat denote the event that triggers an abort in $\mathcal{H}_4$ but not in $\mathcal{H}_3$. In the following we are going to show that $\mathsf{Pr}[\mathsf{cheat} \mid \mathcal{H}_3] \leq \mathsf{negl}(\lambda)$, thus proving the indistinguishability of $\mathcal{H}_3$ and $\mathcal{H}_4$. Assume that the converse is true, then we can construct the following reduction against the discrete logarithm problem (which is implied by the sEUF of ECDSA): On input some $Y^* \in \mathbb{G}$, the reduction guesses a session $j \in [1, q]$ and some index $i \in [1, n]$. The setup algorithm of the j-th session is modified as follows: Y_i is set to be Y^*. Then, for all $\iota \in [i-1, 0]$, the setup samples some $y_\iota \in \mathbb{Z}_q$ and returns $(Y_\iota = Y_{\iota+1} - (y_\iota) \cdot G, Y_{\iota+1}, y_\iota)$. The setup samples a random $y_i \in \mathbb{Z}_q$ and sets $Y_{i+1} = y_i \cdot G$. Then, for $\iota \in [i+1, n-1]$, the setup samples $y_\iota \in \mathbb{Z}_q$ and returns $(Y_\iota, Y_\iota + y_\iota \cdot G, y_\iota)$. The nodes $(U_1, \ldots, U_{n-1})$ are given the corresponding output (except for U_i) and U_n is given

$(Y_{n-1}, \sum_{j=i}^{n-1} y_j)$. If the node U_i is requested to release the lock, the reduction aborts. At some point of the execution the adversary $\mathcal{A}$ outputs some $k^* = (r^*, s^*)$. The reduction parses $s^R = (s', m, \mathsf{pk})$ as the updated state of U_i then checks the following:

1. $\left(\frac{s}{s^*} + y_{i-1}\right) \cdot G = Y^*$
2. $-\left(\frac{s'}{s^*} + y_{i-1}\right) \cdot G = Y^*$

and returns the LHS term of the equation that satisfies the relation.

The reduction is clearly efficient and, whenever j and i are guessed correctly, the reduction does not abort. Since the $\mathbb{G}$ is abelian and the U_i is honest, the distribution induced by the modified setup algorithm is identical to the original to the eyes of the adversary. Recall that cheat happens only in the case where k^* is a valid opening for ℓ_i and the release algorithm is successful on input k^* (if the last condition is not satisfied both $\mathcal{H}_3$ and $\mathcal{H}_4$ abort). Substituting, we have that s' is of the form $\frac{x_0 \cdot x_1 \cdot r_x + H(m)}{r_0 \cdot r_1} = \tilde{s} \cdot y$, where $R' = r_0 \cdot r_1 \cdot Y_{i-1} = (r_x, r_y)$, for some $y \in \mathbb{Z}_q$. Since the release is successful, then it must be the case that $(r_x, \tilde{s})$ is a valid ECDSA signature on the message m_{i-1} (agreed by the two parties in the locking algorithm for ℓ_{i-1}). This implies that $y \cdot G = Y_{i-1}$. As argued in the proof of Lemma 12, if $s^* \neq \tilde{s}$ and $s^* \neq \tilde{s}$, then we have an attacker against the strong unforgeability of the signature scheme. It follows that $s^* = \tilde{s}$ or $s^* = -\tilde{s}$ with all but negligible probability. Substituting we have

$$\begin{aligned}
\left(\frac{s'}{s^*} + y_{i-1}\right) \cdot G &= \left(\frac{\tilde{s} \cdot y}{s^*} + y_{i-1}\right) \cdot G \\
&= \frac{\tilde{s} \cdot y}{s^*} \cdot G + y_{i-1} \cdot G \\
&= y \cdot G + y_{i-1} \cdot G \\
&= Y_{i-1} + y_{i-1} \cdot G \\
&= (Y^* - y_{i-1} \cdot G) + y_{i-1} \cdot G \\
&= Y^*
\end{aligned}$$

which implies that condition (1) holds if $s^* = \tilde{s}$. For the other case

$$\begin{aligned}
-\left(\frac{s'}{s^*} + y_{i-1}\right) \cdot G &= -\left(\frac{\tilde{s} \cdot y}{s^*} + y_{i-1}\right) \cdot G \\
&= -\frac{\tilde{s} \cdot y}{s^*} \cdot G + y_{i-1} \cdot G \\
&= y \cdot G + y_{i-1} \cdot G \\
&= Y_{i-1} + y_{i-1} \cdot G \\
&= (Y^* - y_{i-1} \cdot G) + y_{i-1} \cdot G \\
&= Y^*
\end{aligned}$$

which means that condition (2) is satisfied if $s^* = -\tilde{s}$. Since, by assumption, this happens with probability at least $\frac{1}{q \cdot n \cdot \mathsf{poly}(\lambda)}$ we have a successful attacker against the discrete logarithm problem. This proves our statement. □

This concludes our proof. □

D.6 PCNs from Multi-Hop Locks

In this section we show that AMHLs are sufficient to construct a full-fledged PCN that satisfy the standard security definition from Malavolta et al. [MMSK+17].

Ideal Functionalities. We assume an ideal realization of AMHLs in the form of an ideal functionality $\mathcal{F}_{\mathbb{L}}$ as described in Fig. 5.3. That is, all parties have oracle access to $\mathcal{F}_{\mathbb{L}}$ through the specified interfaces.

Furthermore (same as it was done in [MMSK+17]), we assume the existence of a blockchain B that we model as a trusted append-only bulletin board: The corresponding ideal functionality $\mathcal{F}_{\mathsf{B}}$ maintains B locally and updates it according to the transactions between users. At any point in the execution, anyone can send a distinguished message read to $\mathcal{F}_{\mathsf{B}}$, who sends the whole transcript of B to U. We denote the number of entries of B by $|\mathsf{B}|$. We assume that users can specify arbitrary *contracts*, i.e., transactions in B may be associated with arbitrary conditions which require to be me in order to make the transaction effective. $\mathcal{F}_{\mathsf{B}}$ is entrusted to enforce that a contract is fulfilled before the corresponding transaction is executed.

We model time as the number of entries of the blockchain B, i.e., time t is whenever $|\mathsf{B}| = t$. Note that we can artificially elapse time by adding dummy entries to B and that the current time is available to all parties by simply reading B and counting the number of entries. As discussed before, we assume synchronous communication between users, which is modeled by the functionality $\mathcal{F}_{\mathsf{syn}}$, and secure message transmission channels between users (modeled by $\mathcal{F}_{\mathsf{smt}}$).

Multi-session Extension. A subtlety in the application of the composition theorem is that each call of each ideal functionality assumes to spawn an independent instance. However, the $\mathcal{F}_{\mathbb{L}}$ functionality (described in Fig. 5.3) formally requires a joint state between sessions: The KGen protocols that are used for establishing pairwise links (or channels, respectively) are shared between multiple locking instances which might potentially result in shared keys between the different instances of PrivMuLs that realize payment channels. Consequently, a study of the concrete realization of those KGen protocols is required when arguing about the composition of several locking instances. Composition with joint states is discussed in [CR03], where the authors state a stronger version of the composition theorem (the so-called JUC theorem) which accounts for joint state and randomness across protocol sessions.

In order to satisfy the conditions for the JUC theorem to apply, we must argue that our protocol realizes a stronger ideal functionality $\tilde{\mathcal{F}}_{\mathbb{L}}$ that makes only independent calls to the underlying interfaces (refer to [CR03] for a detailed description). More precisely, this means that we need to argue for each of the previously presented concrete realizations of $\mathcal{F}_{\mathbb{L}}$ that a parallel composition of those protocols – with all instances of the protocol sharing the same KGen protocols and running independently otherwise – realizes the functionality $\tilde{\mathcal{F}}_{\mathbb{L}}$. This is shown in the following lemmas.

Lemma 16. *Let g be a homomorphic one-way function, and let $\widehat{\mathbb{L}_{generic}}^{\mathsf{KGen}}$ be the multi-session extension of the protocol described in Fig. 5.4 using a shared* KGen *algorithm. Then $\widehat{\mathbb{L}_{generic}}^{\mathsf{KGen}}$ UC-realizes the ideal functionality $\tilde{\mathcal{F}}_{\mathbb{L}}$ in the $(\mathcal{F}_{\mathsf{syn}}, \mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{anon}})$-hybrid model.*

Proof. The proof trivially follows from Theorem 2 and the Composition Theorem [Can01] since the KGen protocol never needs to be invoked for realizing $\mathcal{F}_{\mathbb{L}}$ and hence the different copies of $\mathbb{L}_{generic}$ in $\widehat{\mathbb{L}_{generic}}^{\mathsf{KGen}}$ are fully independent. So the joint state is in fact empty. □

Lemma 17. *Let* COM *be a secure commitment scheme, let* NIZK *be a non-interactive zero knowledge proof, and let $\widehat{\mathbb{L}_{schnorr}}^{\mathsf{KGen}}$ be the multi-session extension of the protocol described in Fig. D.2 using a shared* KGen *algorithm realizing $\mathcal{F}_{\mathsf{kgen}}^{\mathsf{schnorr}}$. If Schnorr signatures are strongly existentially unforgeable, then $\widehat{\mathbb{L}_{schnorr}}^{\mathsf{KGen}}$ UC-realizes the ideal functionality $\tilde{\mathcal{F}}_{\mathbb{L}}$ in the $(\mathcal{F}_{\mathsf{kgen}}^{\mathsf{schnorr}}, \mathcal{F}_{\mathsf{syn}}, \mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{anon}})$-hybrid model.*

Proof. It is easy to see that the $\mathcal{F}_{\mathsf{kgen}}^{\mathsf{schnorr}}$ functionality itself is stateless and, therefore, consecutive invocations of $\mathcal{F}_{\mathsf{kgen}}^{\mathsf{schnorr}}$ are indistinguishable from the invocation of fresh instances of the functionality. Hence, for multiple protocols, it is identical to query the same $\mathcal{F}_{\mathsf{kgen}}^{\mathsf{schnorr}}$ instance or to work on independent copies (and the same property carries over to protocols realizing this functionality). As a consequence $\widehat{\mathbb{L}_{schnorr}}^{\mathsf{KGen}}$ is indistinguishable from the multi-session extension of $\mathbb{L}_{schnorr}$ using independent KGen copies that realize $\mathcal{F}_{\mathsf{kgen}}^{\mathsf{schnorr}}$. So the claim trivially follows from Theorem 7 and the Composition Theorem [Can01]. □

Lemma 18. *Let* COM *be a secure commitment scheme and let* NIZK *be a non-interactive zero knowledge proof, and let $\widehat{\mathbb{L}_{ecdsa}}^{\mathsf{KGen}}$ be the multi-session extension of the protocol described in Fig. 5.5 using a shared* KGen *algorithm realizing $ideal_{\mathsf{kgen}}^{\mathsf{ECDSA}}$. If ECDSA signatures are strongly existentially unforgeable and Paillier encryption is* ecCPA *secure, and* KGen *then the construction in Fig. 5.5 UC-realizes the ideal functionality $\tilde{\mathcal{F}}_{\mathbb{L}}$ in the $(\mathcal{F}_{\mathsf{kgen}}^{\mathsf{ECDSA}}, \mathcal{F}_{\mathsf{syn}}, \mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{anon}})$-hybrid model.*

Proof. As $\mathcal{F}_{\mathsf{kgen}}^{\mathsf{ECDSA}}$ satisfies the same independence property as $\mathcal{F}_{\mathsf{kgen}}^{\mathsf{schnorr}}$, the same argument as for Lemma 17 applies. □

System Assumptions. We assume that every user in the PCN is aware of the complete network topology, that is, the set of all users and the existence of a payment channel between every pair of users. We further assume that the sender of a payment chooses a payment path to the receiver according to her own criteria.

The current value on each payment channel is not published but instead kept locally by the users sharing a payment channel. The two users U_0 and U_1 are assumed to maintain locally the capacity of their channel, denoted by $\mathsf{cap}(U_0, U_1)$. We further assume that every user is aware of the payment fees charged by each other user in the PCN. For ease of exposition, we define the predicate $\mathsf{fee}(U_i)$ to return the fee charged by the user U_i. We assume that pairs of users sharing a

payment channel communicate through secure and authenticated channels (such as TLS), which is easy to implement given that every user is uniquely identified by a public key.

Our System. In the following, we describe the three operations (open channel, close channel, and payment) that constitute the core of our system. For the sake of simplicity, we restrict each pair of users to at most one channel. However, our construction can be easily extended to support multiple channels per pair.

OPEN CHANNEL. The open channel protocol generates a new payment channel between users U_1 and U_2. The user U_1 invokes $\mathcal{F}_{\mathbb{L}}$ on input (U_2, L), depending on the direction of the channel, which returns the users' identifiers (U_1, U_2) if the operation was successful. Then the users create an initial blockchain deposit that includes the following information: Their addresses, the initial capacity of the channel, the channel timeout, and the fee charged to use the channel agreed beforehand between both users. After the deposit has been successfully added to the blockchain, the operation returns 1. If any of the previous steps is not carried out as defined, the operation returns 0.

CLOSE CHANNEL. The close channel protocol is run by two users U_1 and U_2 sharing an open payment channel to close it at the state defined by v and accordingly update their bitcoin balances in the Bitcoin blockchain. From this point on, U_1 and U_2 ignore all the requests from $\mathcal{F}_{\mathbb{L}}$ relative to their link.

PAYMENT. A payment operation transfers a value v from a sender (U_0) to a receiver (U_{n+1}) through a path of open payment channels between them $(U_0, \ldots, U_{n+1})$. The sender (Algorithm 1) first computes the cost of sending v coins to the receiver as $v_1 := v + \sum_{i=1}^{n} \mathsf{fee}(U_i)$, and the corresponding cost at each of the intermediate hops in the payment path. Then it setups up a AMHL by calling the ideal functionality $\mathcal{F}_{\mathbb{L}}$ on the set of identifiers of the intermediate users. Finally, it sends each user the corresponding value to be transferred and a timeout information t_i.

Each intermediate user (Algorithm 3) checks whether the capacity of the channel is high enough to support the transfer of the coins and whether the timeouts give by the sender are consistent, i.e., $t_{i+1} = t_i - \Delta$ for some fixed Δ. Starting from (U_0, U_1), each pair of users query the ideal functionality $\mathcal{F}_{\mathbb{L}}$ on the **Lock** interface using the *lid* received in the previous phase. If the ideal functionality signals to proceed, then the two users establish a contract specified in the following.

contract(Alice, Bob, *lid*, x, t)

1) If GetStatus(*lid*) = Rel before t days,
then Alice pays Bob x coins.
2) If t elapse, then Alice gets back x coins.

The contract is authenticated by both users and can be uploaded to B by either of them at any time. If every user in the path locks the corresponding *lid*, eventually the receiver (Algorithm 2) is reached. U_{n+1} checks whether the transacted value is what it expects and whether the latest timeout t_{n+1} is well-formed. If both conditions hold, the receiver releases the lock lid_n by

querying the ideal functionality. This triggers a cascade of release calls in the path from the sender to the receiver, thereby enabling the left user in the link to pull the payment (using the previously established contract). If for some reason, one of the intermediate links is not released, then all of the previous contracts are voided after the corresponding timeout.

Algorithm 1: Payment routine for the sender

Input : $(U_0, \ldots, U_{n+1}, v)$
$v_1 := v + \sum_{i=1}^{n} \mathsf{fee}(U_i)$
if $v_1 \leq \mathsf{cap}(U_0, U_1)$ **then**
  query $\mathcal{F}_{\mathbb{L}}$ **on Setup**$(U_0, \ldots, U_{n+1})$
  $\mathcal{F}_{\mathbb{L}}$ **returns** $(\bot, \mathit{lid}_0, \bot, U_1, \mathsf{Init})$
  $\mathsf{cap}(U_0, U_1) := \mathsf{cap}(U_0, U_1) - v_1$
  $t_0 := t_{\text{now}} + \Delta \cdot n$
  forall $i \in \{1, \ldots, n\}$
    $v_i := v_1 - \sum_{j=1}^{i-1} \mathsf{fee}(U_j)$
    $t_i := t_{i-1} - \Delta$
    send $((U_{i-1}, U_{i+1}, v_{i+1}, t_i, t_{i+1}), \mathsf{fwd})$ **to** U_i
  end for
  send (U_n, v_{n+1}, t_{n+1}) **to** U_{n+1}
  query $\mathcal{F}_{\mathbb{L}}$ **on Lock**(lid_0)
  if $\mathcal{F}_{\mathbb{L}}$ **returns** $(\mathit{lid}_0, \mathsf{Lock})$
    $\mathsf{contract}(U_0, U_1, \mathit{lid}_0, v_1, t_1)$
  else
    abort
  end if else
  abort
end if

Algorithm 2: Payment routine for the receiver

Input : $(U_n, v_{n+1}, t_{n+1}, v)$
$\mathcal{F}_{\mathbb{L}}$ **returns** $(\mathit{lid}_n, \bot, U_n, \bot, \mathsf{Init})$
if $(t_{n+1} > t_{\text{now}} + \Delta) \wedge (v_{n+1} = v) \wedge$ (**GetStatus**$(\mathit{lid}_n) = \mathsf{Lock}$) **then**
  query $\mathcal{F}_{\mathbb{L}}$ **on Release**(lid_n)
  send ok **to** U_n
else
  send $\bot$ **to** U_n
end if

Analysis. In the following we argue that the system as described above ideally realizes the functionality $\mathcal{F}_{PCN}$ as defined in [MMSK+17], assuming oracle access to $\mathcal{F}_{\mathbb{L}}$, $\mathcal{F}_{\mathsf{B}}$, and $\mathcal{F}_{\mathsf{syn}}$.

Algorithm 3: Payment routine for the i-th intermediate user

Input : $(m, decision)$
if $decision = \mathsf{fwd}$ **then**
  parse m **as** $(U_{i-1}, U_{i+1}, v_{i+1}, t_i, t_{i+1})$
  $\mathcal{F}_{\mathbb{L}}$ **returns** $(lid_{i-1}, lid_i, U_{i-1}, U_{i+1}, \mathsf{Init})$
  if $(v_{i+1} \leq \mathsf{cap}(U_i, U_{i+1})) \wedge (t_{i+1} = t_i - \Delta) \wedge$ (**GetStatus(**lid_{i-1}**)** $= \mathsf{Lock}$) **then**
    $\mathsf{cap}(U_i, U_{i+1}) := \mathsf{cap}(U_i, U_{i+1}) - v_{i+1}$
    query $\mathcal{F}_{\mathbb{L}}$ **on Lock(**lid_i**)**
    if $\mathcal{F}_{\mathbb{L}}$ **returns** (lid_i, Lock)
      $\mathsf{contract}(U_i, U_{i+1}, lid_i, v_{i+1}, t_{i+1})$
    else
      send $\perp$ **to** U_{i-1}
    end if
  else
    send $\perp$ **to** U_{i-1}
else if $decision = \perp$ **then**
  $\mathsf{cap}(U_i, U_{i+1}) := \mathsf{cap}(U_i, U_{i+1}) + v_{i+1}$
  send $\perp$ **to** U_{i-1}
else if $(decision = \mathsf{ok}) \wedge$ **GetStatus(**lid_i**)** $= \mathsf{Rel}$ **then**
  query $\mathcal{F}_{\mathbb{L}}$ **on Release(**lid_{i-1}**)**
  send ok **to** U_{i-1}
else
  send $\perp$ **to** U_{i-1}
end if

Theorem 8. *The system described above UC-realizes $\mathcal{F}_{PCN}$ (as defined in [MMSK+17]) in the $(\mathcal{F}_{\mathbb{L}}, \mathcal{F}_{\mathsf{B}}, \mathcal{F}_{\mathsf{syn}}, \mathcal{F}_{\mathsf{smt}})$-hybrid model.*

Proof. The proof consists of the observation that the ideal functionality $\mathcal{F}_{\mathbb{L}}$ enforces balance security and satisfies relationship anonymity (as defined in [MMSK+17]). A subtlety is that now all users have access to a **GetStatus** interface, and they might be able to query the functionality on a certain *lid* and learn its status even when they are not involved in the generation of such a lock. However, one can easily show that this happens only with negligible probability since it requires guessing *lid*, which is a string sampled uniformly at random. It is also easy to see that $\mathcal{F}_{\mathbb{L}}$ does not allow one to perform wormhole attacks by construction. What is left to be shown is that the rest of the information exchanged by the machines does not break any of these properties. Note that the only information that is sent outside $\mathcal{F}_{\mathbb{L}}$ consists of user identifiers, timeouts, and values to lock. The first identifiers are already known by the intermediate users, whereas the rest of the items are precisely chosen as described in $\mathcal{F}_{PCN}$. Note that it is sufficient here to argue about the individual copies of $\mathcal{F}_{\mathbb{L}}$ in isolation by the JUC theorem [CR03]. As we showed, the multi-session extended ideal functionality $\tilde{\mathcal{F}}_{\mathbb{L}}$ is realized by our instantiations, and, therefore, the

JUC theorem allows us to complete the analysis assuming independent copies of $\mathcal{F}_{\mathbb{L}}$ running in parallel. □