FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Theoretische und praktische Smart Contracts - Realisierung eines Investmentfonds

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Jakob Felix Schneider
Matrikelnummer 01528502

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: Univ.Prof. Dr. Matteo Maffei
Mitwirkung: Univ.Ass. Clara Schneidewind BSc, Projektass. Ilya Grishchenko MSc

Wien, 25.07.2018 ____________________ ____________________

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Erklärung zur Verfassung der Arbeit

Jakob Felix Schneider
Reznicekgasse 10/9
1090 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, der 25.07.18 ________________________

Abstract. In the last years, the blockchain technology has been adopted by many traditional fields (such as the financial sector) for reinventing existing applications and giving rise to new use cases. However, during this period of rapid progress the spotlight remained on fast product development, which has led to many security issues throughout the last years resulting in hundreds of millions of USD stolen or lost. Nearly all current public blockchains offer no formal semantics or formal frameworks for verifying smart contract code. This thesis presents a novel semantics for smart contract interactions and a state-of-the-art implementation of a smart-contract-based investment fund for the Ethereum blockchain. The focus lies on connecting a formal semantics for smart contracts to the actual business use case of an investment fund. Specifically, the semantics introduced provides a novel approach by targeting smart contract interactions rather than single smart contract executions. The blockchain is represented by a global state on which complex transaction can be modelled using a big-step semantics. The fully-fledged investment fund ERCFund implemented in the course of this thesis makes it possible to invest in an actively managed portfolio of ERC20-Tokens and Ether by introducing an on-demand minted and burned token as the medium for shares in the fund. Moreover, the software supports many advanced features, such as cold-wallet support and multi-signature protection with off-chain signing. We show that the novel semantics introduced is applicable in a real business setting by adapting and proving an integral security feature for a building block of our investment fund.

Abstract. Innerhalb der letzten Jahre wurden Blockchain-Technologien in vielen traditionellen Sektoren (z.B. im Finanzbereich) adaptiert. Dadurch wurden existierende Systeme neu überdacht und ganzheitlich neuartige Systeme erfunden. Allerdings blieb während dieser Zeit des raschen Fortschritts der Fokus stets auf der schnellen Produktentwicklung, was zu vielen Sicherheitsproblemen geführt hat durch die mehrere hundert Millionen von USD gestohlen oder verloren wurden. Nahezu alle derzeitig öffentlichen Blockchains bieten keine formale Semantik oder formales Framework zur Verifizierung von Smart-Contract-Code. Diese Arbeit präsentiert eine neuartige Semantik für Smart-Contract-Interaktionen und eine state-of-the-art Implementierung eines smart-contract-basierenden Investmentfonds für die Ethereum Blockchain. Der Fokus liegt darauf, eine formale Semantik für Smart Contracts mit dem tatsächlichen, wirtschaftlichen Anwendungsfall eines Investmentfonds zu verbinden. Genauer gesagt, die eingeführte Semantik bietet einen neuartigen Denkansatz dadurch, dass sie Smart Contract Interaktionen und nicht einzelne Smart-Contract-Ausführungen in den Mittelpunkt stellt. Die Blockchain wird durch einen Global State repräsentiert, auf welchem komplexe Transaktionen durch eine Big-Step-Semantik modelliert werden können. Der im Laufe dieser Arbeit entwickelte Investmentfonds ERCFund macht es möglich, in ein aktiv verwaltetes Portfolio von ERC20-Tokens und Ether zu investieren. Dies wird realisiert durch Tokens, welche als Anteil des Investmentfonds genutzt werden und je nach Bedarf gemünzt und vernichtet werden können. Außerdem unterstützt die Software mehrere fortgeschrittene Funktionen, wie z.B. Cold-Wallet-Support und Multi-Signature-Schutz durch Off-Chain-Signaturen. Wir zeigen, dass die eingeführte Semantik in einer realen, geschäftlichen Situation anwendbar ist, indem wir eine wesentliche Sicherheitsfunktion des Investmentfonds adaptieren und formal beweisen.

Table of Contents

1 Introduction 3

1.1 Related scientific work 4

1.2 Related products on the market 5

2 Blockchain technology 7

2.1 Fundamentals 7

2.2 Smart contract application blockchains 12

2.3 Blockchain related attack vectors 13

3 Designing an investment fund for the blockchain 18

3.1 Funding 18

3.2 Tokens as shares 19

3.3 Managing multiple currencies 22

3.4 Possible fee structures 23

3.5 Security features 24

4 Blockchain semantics 26

4.1 Design choices 26

4.2 Preliminaries 28

4.3 Semantics 30

5 Modelling smart contracts 41

6 Investment fund implementation 50

6.1 Architecture 51

6.2 Implementation 54

7 Conclusion 65

1 Introduction

Since Bitcoin's inception eight years ago, hundreds of other cryptocurrencies were created and are being publicly traded. Within these eight years, the cryptocurrency sector has grown to a market cap of around USD 835 B [7]. Furthermore in the six months before that, the market cap has grown more than eightfold. This unprecedented growth has spawned questions about cryptocurrency as an investment and has attracted many traditional investors via traditional investment funds (e.g., Grayscale Bitcoin Trust [13]).
Cryptocurrencies are often seen as an investment vehicle like no other, on one hand they can be used as global currency for e.g., payment purposes on the other hand they can also act as a share-holding mechanism for companies and replace current technologies such as the stock market. The many different use cases of cryptocurrencies and the technical entry barrier of understanding their inherent technological value create a whole new investment sector which is still being explored.
Existing cryptocurrency funds, like aforementioned Grayscale, have implemented simple funds. An interested party can purchase shares which represent a set of assets. Security and warranty are defined by the strict legal framework on which traditional funds are built upon.
However the blockchain allows for arguably stricter security and warranty with its immutability. Calls for a modern and technologically more fitting implementation of cryptocurrency funds have been expressed by companies such as Crypto20 [61] and TaaS.fund [41] who currently manage cryptocurrency assets worth tens of millions of USD. While companies have used blockchain based technologies to implement parts of an investment fund, for example profit payout, no fully-fledged fund structure on the blockchain has been published to our knowledge.

Besides Bitcoin, arguably more advanced blockchains have been developed and adopted. The second biggest player, judging by market capitalization, is Ethereum [40] which describes itself as a ”next-generation smart contract and decentralized application platform”. Smart contracts are an advanced blockchain feature and enable programs and applications to run directly on the blockchain. These applications are then visible and callable for everyone and provide full transparency about their functionality and past executions.

However one of the biggest problems blockchain ecosystems with smart contracts like Ethereum are facing right now is smart contract security. While the Ethereum blockchain itself did not have a significant software bug yet, many user-created smart contracts had serious bugs. One of the earliest was the theDAO bug [67], which resulted in around USD 60M stolen and ultimately triggered a controversial hard fork to restore the funds.
Smart contract bugs are not the exception and are not a sign of inexperience: The company Parity Technologies has run into two successive, serious bugs related to one of their products, a smart-contract-based wallet. The code of the wallet was at least partially written and audited by the co-founder and mastermind of Ethereum itself, Gavin Wood. All in all at least 150,000 and 500,000 Ether respectively were lost (around USD 180M at the time the bugs occurred). It is clear that formal verification is direly needed

in the blockchain world, where a small piece of code can handle millions of USD on a regular basis.
By developing a fund structure and formally verifying the integrity of all parts, it would be possible to revolutionize fund security and integrity for and via cryptocurrency.

1.1 Related scientific work

The Ethereum Virtual Machine (EVM), which runs smart contracts translated into the EVM bytecode, currently has no official semantics. An initiative was started by Hildebrandt et al. [50] to define an EVM semantics in the $\mathbb{K}$ framework [64]. They utilize the framework to create a fully executable semantics and test it against the Ethereum test suite. However, the semantics currently requires a large amount of user input, such as specifying loop invariants manually which is infeasible in some cases.
Grishchenko et al. [48] introduce a complete small-step semantics of the EVM bytecode which they validate against the Ethereum test suite. Furthermore they introduce several security properties for smart contracts and uncover flaws in existing semantics and verification tools.
While we do not introduce a semantics for the EVM bytecode in this thesis, it is a critical part of smart contract executions and could potentially introduce unavoidable security issues.

The call for safe smart contracts is widely spread, Luu et al.[55] discuss security flaws found in many smart contracts on the Ethereum blockchain. They developed *Oyente* which is an automated tool for discovering vulnerabilities in smart contracts. It flagged around half of all existing, deployed smart contracts as potentially vulnerable. Additionally they discuss an operational semantics of Ethereum describing its transactions and offer a new semantics to address security flaws they discovered.
Delmolino et al.[43] describe their lessons learned from writing smart contracts, how to avoid common pitfalls and implement best practices for smart contracts.
First steps into the direction of formally verified smart contracts are being made: Bhargavan et al.[34] have defined a limited subset of the smart contract programming language Solidity. They translate this subset into F* and formally verify smart contracts with F*'s proof assistant. However, the tool only works with a fraction of available contracts and they do not provide a sufficient semantic framework.
Grossman et al. [49] wrote an extensive paper regarding callbacks in smart contracts. They analyze a common smart contract vulnerability often termed as *reentrancy* which is responsible for the most famous smart contract hack, theDAO hack. In the paper the notion of an *Effectively Callback Free Object* is introduced which does restrict the functionality of callbacks but certifies that any state reached with callbacks can also be reached without using callbacks. Additionally, Grossman et al. analyzed more than a 100 million smart contract executions for potential reentrancy vulnerabilities.
In [52] a framework for analyzing safety properties of smart contracts is presented by Kalra et al. They leverage abstract interpretation, symbolic model checking and the power of constrained Horn clauses to provide a tool for quickly verifying Ethereum smart contracts for safety. With it, they have evaluated around 22,000 smart contracts containing 0.5 billion USD and have found out that around 95% of them are vulnerable.

Scilla is a smart contract language designed for formally verifying smart contracts [63]. Specifically, Scilla acts as an intermediate translation target for a high-level language like Solidity. Uniquely, the communication and programming part of a smart contract are completely separated as opposed to in other common languages. Similarly to the semantics in this thesis, the smart contracts are limited to using tail-calls to call further smart contracts. Scilla has been ported to the proof assistant Coq to verify properties such as liveness and safety.

So far most of the research towards formalizing blockchain semantics focuses either on the EVM byte-code level or on the analysis of isolated contracts. By introducing an abstracted semantics focusing on smart contract interaction we make it possible to describe transactions spanning over several smart contracts.

Currently there are little to no blockchains which utilize formal verification. However, an initiative worth mentioning is Cardano which is currently developed by a team of academics consisting of university professors and postdoctoral researchers. They are working on Ouroboros [53] which is a provably secure proof-of-stake blockchain consensus protocol. Proof-of-stake is an alternative to the currently wide-spread proof-of-work which requires high computing power in order to confirm a transaction on the blockchain. Proof-of-stake on the other hand uses a stake of currency as a collateral for confirming transactions. Furthermore Ouroboros is planned to be implemented for their cryptocurrency called *Ada* which is currently one of the ten biggest cryptocurriences in market capitalization with multiple billion USD [7].

1.2 Related products on the market

Nowadays many different blockchain technologies exist which are capable of running quasi-turing-complete programs, e.g., Ethereum [40], NEO [17]. Many traditional structures are reimagined and implemented decentralized, investment funds are not an exception.

Currently a wide variety of investment funds, which at least partially operate on the blockchain, is available. For example the Crypto20 fund [61] is a tokenized cryptocurrency index fund: It acts as an index fund in the way that it always invests in the 20 cryptocurrencies with the highest market capitalization. It uses smart contracts to manage some of its parts. Shares are implemented via smart contracts (Ethereum ERC20-Tokens), in addition it uses a smart contract which enables payout by trading shares based on the current net asset value (NAV). This ensures that the price of a share cannot drop under the NAV.

Another example is the TaaS.fund[41] which is one of the first cryptocurrency funds actively managing funds with a team of traders. It uses a smart contract to payout profits of the fund quarterly to all wallets holding shares of the fund.

The Melon protocol is an autonomous blockchain protocol, designed to allow for the purposes of cryptocurrency asset management [66]. It uses multiple existing technologies such as IPFS and Ethereum smart contracts to enable users to create their own investment funds directly on the blockchain. Melon is unique because to our knowledge it is the only fund management system which is truly completely decentralized

and purely built on the blockchain.
Because of its popularity we still want to include ICONOMI which is a digital asset management platform [70]. However, ICONOMI does not fall into the same class as the other funds mentioned above because it is not directly built using blockchain technology but rather is similar to traditional investment platforms. ICONOMI offers a variety of cryptocurrency funds for their customers which are managed by a few selected partners. Interestingly, it is one of the few companies which has a token representing equity in the company, as it provides owners with a portion of the firm's profit.

2 Blockchain technology

2.1 Fundamentals

The term Blockchain describes the idea of a continuously chained list of information bundled in the form of blocks. This kind of structure is actually not a novelty of cryptocurrencies but was first coined by Ralph Merkle [57] and is known as a Merkle Tree or Hash Tree. Every block in a blockchain is linked to the one before it by using a cryptographic hash for identification, except for the root/initial block. For cryptocurrencies these blocks contain information about transactions which were confirmed by publishing the block, but can also contain other information depending on the implementation. By design the blockchain consequently acts as a cryptographic ledger for a cryptocurrency.

The blockchain as we know it today was invented by Satoshi Nakamoto who created the first, fully-fledged, decentralized cryptocurrency Bitcoin [58] in 2008. Bitcoin allows for participants of the blockchain, who are identified by a quasi-unique hash, to transfer currency (Bitcoins) to other participants. By offering a public, decentralized ledger where every new entry has to be confirmed by a large number of miners (participants who confirm transactions) representing at least 51% of the network, it creates a secure, transparent payment system which has a high fault tolerance.

In this section we give an introduction to blockchain technology mainly based on Ethereum. Many newer blockchains have some different features, but taking a look at the most popular cryptocurrency which supports smart contracts gives a good foundation.

Decentralized public ledger We just mentioned the decentralized public ledger is a chain of blocks. This chain includes every past, legitimate transaction from the launch of a cryptocurrency. But we did not yet cover how exactly this decentralized public ledger is used to ensure consensus and safety around the globe.

The blockchain is available for anyone to see via a network of public nodes. This network is a peer-to-peer network where new nodes typically first discover a node and then add all trusted nodes from the discovered node. If we speak of *nodes* we typically mean *full nodes* which means they save a complete copy of the blockchain locally. New nodes can download their copy, simultaneously verify it, and use it to verify future transactions or to just act as a relay point. Many networks nowadays introduce node types which do not require the whole blockchain to be saved, commonly called *light nodes*. This is enabled through saving only relevant parts of the blockchain by consulting full nodes about the current consensus. Light nodes cannot offer full verification of the blockchain but are in many cases sufficient and enable devices with low storage capacity to act as nodes. At the time of writing running a full Ethereum node using geth [21] uses upwards of 50 Gigabytes of storage.

In Figure 1 we can see a simplified blockchain. The green blocks signalize the main chain which was agreed on with a consensus algorithm. This unique valid chain of blocks contains all past transactions of the blockchain and can be used to derive the

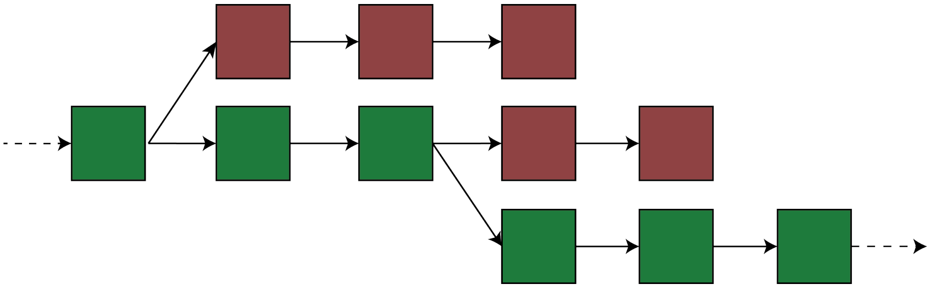

Fig. 1: Simplified visualization of a part of a blockchain.

current state of the blockchain (e.g. balances of addresses). The main chain is also the longest chain of valid blocks.
After a new block is successfully mined, the node which mined the block will publish it across the network. Other nodes then verify the correctness of the block and if the block turns out to be valid, they will accept it as part of the main chain and start working on the next one, because only new blocks offer monetary rewards.
Possible attack vectors to the public ledger are discussed in Subsection 2.3, but up to now the Bitcoin and Ethereum public ledger did not suffer any successful large scale attack to our knowledge.

The cryptographic ledger keeps track of all past transactions and therefore all relevant information of the global state. In its simplest form this means the balance of all participants. Participants on blockchains are accounts identified by a unique address (a long hex string) which can only be accessed by their owners (see Section 2.1). The blockchain is a list of past transactions and by observing these transactions a definitive balance of any given address can be determined. So if for example an address would try to send more currency than it currently possesses, it can be determined that the transaction is invalid. It is crucial to understand that no value is directly *held* by any address or wallet. The balance of an address is derived by its past transactions.
In newer blockchains, such as Ethereum, addresses are not exclusively controlled by external parties, but can also be associated to self-executing programs (smart contracts; covered in Subsection 2.2). The public ledger keeps track if a smart contract is deployed at an address. If so, all functions of the smart contract can be accessed/executed via transactions. We expand on this in Subsection 2.2.

Accounts Participants of the blockchain, commonly called accounts, are the owners/holders of an address. We talked about how addresses can hold value and send transactions and how they are publicly visible. This opens the question how it is possible to ensure authenticity of the owner of an account.

The address is a 40 byte value derived from the public key and consequently the private key. The procedure is illustrated in Figure 2. A private key is a randomly chosen 64 bytes hexadecimal value which is transformed with the Elliptic Curve Digital Signature Algorithm (ECDSA) [51] using the secp256k1-curve [62] to a 128 bytes hexadecimal

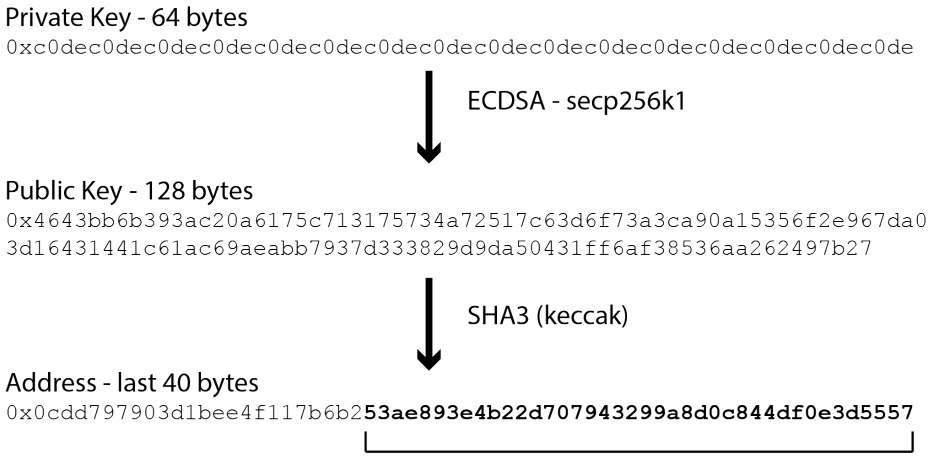
Private Key - 64 bytes
0xc0dec0dec0dec0dec0dec0dec0dec0dec0dec0dec0dec0dec0dec0dec0dec0de
ECDSA - secp256k1
Public Key - 128 bytes
0x4643bb6b393ac20a6175c713175734a72517c63d6f73a3ca90a15356f2e967da0
3d16431441c61ac69aeabb7937d333829d9da50431ff6af38536aa262497b27
SHA3 (keccak)
Address - last 40 bytes
0x0cdd797903d1bee4f117b6b2**53ae893e4b22d707943299a8d0c844df0e3d5557**

Fig. 2: Derivation from private key to public key to address. Example adapted from [42].

value. This value is the public key, which is misleading in this case as it is not publicly visible. rather it is hashed with the standard SHA3 (Keccak) [46]. From the resulting hash the last 40 bytes are taken and used as the address of the account.
The private key is used to authorize transactions from the address by signing the transaction details. This signature is needed to successfully publish a transaction (otherwise it will be rejected by honest nodes) which provides a secure layer and makes it impossible for anyone who does not know the private key to publish a transaction from the address (as long as the ECDSA remains unbroken).

Transactions In the previous section we described how the public ledger is a collection of transactions which when subsequently executed provides the current global state of the blockchain. In a sense, **the blockchain is a transaction based state machine**, with transactions changing the state and the global decentralized ledger keeping track of changes. A transaction gives instructions on how to apply changes to the global state when executed. For example, if you would send currency to someone else, you would do so via a transaction.

nonce: `3`	gasprice: `1`	gas limit: `21,000`	to: `0xdeadbeef...`	value: `1,000`	data: `0x42fab21...`

Fig. 3: Input parameters for an Ethereum transaction.

In Figure 3 the input parameters for an Ethereum transaction are highlighted. Additionally, the input parameters of a transaction have to be signed by the private key of the address to authenticate the sender. This happens with the ECDSA which produces the according signature which is simply added to the transaction payload.

Blocks Transactions are saved on the blockchain in the form of blocks, a block is nothing more than a certain amount of transactions grouped up. Blocks are used to order the execution of transactions. The order of transactions within one block is mostly arbitrary and is decided by miners, but all transactions of the next block are strictly afterwards in the timeline. The fact that miners can decide the ordering and the exact publishing time of a block can introduce vulnerabilities if code depends on the order of transactions or can be frontrun. More on this in Subsection 2.3.
Block size and block time (how often new blocks are mined) vary greatly from blockchain to blockchain. For example Bitcoin has an average blocktime of 10 minutes and a block size of around 1 Megabyte which fits one to two thousand transactions [1]. Ethereum only has an average blocktime of 15 seconds with a block size of around 20 Kilobytes which usually fits between 70 and 200 transactions depending on the cost of these transactions [11].

Mining or transaction confirmations Mining was mentioned several times in the last section already as a mechanism to confirm transactions and group them into a block. Verifying transactions is a delicate part of the blockchain because it needs to be bound to some kind of resource or identity. By simply allowing confirmations from anyone it would be easily possible to flood the blockchain with malicious confirmations by organizations with high internet throughput, similarly to a Denial of Service (DoS) attack.

Proof of Work The early blockchains, such as Bitcoin, solved this problem by creating a concept called Proof of Work (PoW). It describes the act of proving that a real machine is behind the confirmation by requiring miners to solve a difficult mathematical problem which requires a lot of processing power. With Bitcoin, all miners try to solve a mathematical problem for a block they want to verify and publish, whoever finds the correct answer to this problem gets to publish their block and receive a mining reward. Specifically, they need to find a random nonce which combined with the block hash equals to a given hash.
This is a working concept, as anyone who would want to publish a (lasting) malicious transaction would need 51% of the network's computing power. However it is possible for a malicious or faulty block to be mined by sheer coincidence (even if the attacker only has 0.1% computer power). It is unlikely for the same attacker to be able to mine a second or even third block afterwards, because consequently other miners and nodes will identify the block as faulty and will not continue to mine on it and will not broadcast it across the network.

There are several commonly voiced issues with PoW mining:

- Energy consumption: PoW consumes an enormous amount of energy by solving *practically useless* problems with the only purpose of ensuring energy is actually consumed. Currently Bitcoin alone uses an estimated amount of 63 TWh yearly which is around the amount the whole country of Switzerland consumes yearly [2]. *Wasting* this amount of energy is argued by many to be environmentally irresponsible.

- ASICs: ASIC stands for Application-Specific Integrated Circuit and is a circuit that is built specifically for one use-case. Bitcoin mining is dominated by ASIC machines built specifically to solve aforementioned hash problems efficiently. They can achieve a much higher efficiency with lower energy consumptions which makes it hardly profitable to mine Bitcoin for anyone owning a normal computer. It is argued that ASICs are *centralizing* Bitcoin by introducing a high entry barrier for miners.
- Mining pools: For widely adopted PoW cryptocurrencies it is the norm to mine not independently but as part of a mining pool. The reason for this is that only one miner per block receives the whole reward and it is completely random who this miner is. Consequently you could mine for months alone and not receive any reward for it. Mining pools bundle up many miners and if one miner *wins the lottery* the reward is split based on mining power. This is going against one of the core values of cryptocurrency: Decentralization. Mining pools have reached dangerous levels of centralization, to the point where for Bitcoin if the three biggest mining pools would group up they could take over the blockchain by controlling more than 51% of the mining power [3]. It is also important to note that this problem is not inherent to PoW but rather a consequence of the design of block rewards.

Proof of Stake Proof of Stake (PoS) is a newer concept which uses a different resource to prove commitment to the blockchain: Cryptocurrency itself. Instead of proving processing power miners have to stake a certain amount of currency to confirm transactions on network nodes. Staking currency usually makes it unusable for a certain amount and acts like a bond. Miners get a reward for confirming honest blocks by simply owning cryptocurrency, a concept that seems familiar: Gaining interest by staking your money at a bank. Confirming a malicious block and therefore being dishonest has a direct penalty associated to it. This penalty is essential because otherwise there would be no discouragement for it, while in PoW you get indirectly punished by burning energy. PoS is already used by many blockchains such as NEO [18] and is planned to be implemented by Ethereum as part of the Casper protocol update [71].

PoS solves many issues of PoW, like the energy consumption, but might not be the end to all problems. For most blockchains it is not possible to stake small amounts which creates an entry barrier (although smaller than ASIC's). For reference Ethereum's PoS protocol will initially have a minimum stake of 1,000 Ether [36] which is around USD 600,000 at the time of writing and aims to reach a minimum stake of 32 Ether (USD 20,000) with sharding technology [37]. A common misconception of PoS is that it gives more power to the rich compared to PoW. It currently *only* takes estimated USD 5 billion dollars to buy enough computing power to start a 51% attack on the Bitcoin blockchain [5], although it has a market cap of USD 150 billion dollars. If Bitcoin would use PoS it would likely cost significantly more to gain 51%.

2.2 Smart contract application blockchains

Bitcoin allows for transactions to transfer value and enhances them with a simple scripting language for some advanced features such as multi-signature transactions. Participants of these kind of blockchains are always external accounts and transactions are directly initiated by them. Smart contract application blockchains introduce another type of participant to the blockchain: Contracts. These are computer programs directly published and saved on the blockchain by an external account. Anyone can see and interact with smart contracts by sending transactions to their corresponding address.
Many modern blockchains, such as Ethereum, implement a *quasi-turing-complete* language which is only limited by external transaction costs. Smart contracts make it possible to run any program decentralized on the blockchain and consequently enable a new form of computing. Smart contracts enable many use cases e.g., share-holding [60], advanced money exchanges [69], identity management [32], and off-chain transactions [31]. For example, it is possible to implement a decentralized cryptocurrency exchange [69] with an absolute medium of trust: The code. Anyone can look at exactly what kind of program they are using and nothing is left uncertain. All past executions of smart contracts are part of the blockchain.

```
 1  contract SimpleSavingsWallet {
 2
 3    function SimpleSavingsWallet() public {}
 4
 5    // Wallet can receive funds.
 6    function () public payable {}
 7
 8    // Wallet can send funds.
 9    function sendTo(address payee, uint256 amount) public {
10      require(payee != 0 && payee != address(this));
11      require(amount > 0);
12      payee.transfer(amount);
13    }
14  }
```

Fig. 4: Small example of a wallet smart contract. Adapted from [59].

In Figure 4 a simple example of a smart contract in the programming language Solidity [28] for the Ethereum blockchain is given. Solidity is the currently most used programming language for smart contracts on the Ethereum blockchain, additionally the investment fund presented in Section 6 is purely written in Solidity. We can observe that smart contracts programming languages use known concepts out of object-oriented programming to define their functionality, such as constructors and class functions. This example depicts a smart contract which can receive and send Ether and is by default usable by any party on the blockchain. In line 6 the contract specifies that its default function is `payable`, which means that the contract can receive standard payments. In line 9 a transfer function is defined, which first checks if the payee is a valid address and if the amount is positive. Afterwards the amount is sent to the payee.

Smart contract application platforms extend the global state of a simpler blockchain, such as Bitcoin, with additional storage capacities in order to support deployment and persistence of smart contracts. While cryptocurrencies like Bitcoin save not much more than balances and past transactions, in Ethereum every address can additionally save the bytecode of a contract. In the case one is deployed there, and has a dedicated storage for a contract. This storage is used to ensure that contracts are able to be stateful. From a traditional software engineering standpoint this means contracts can have class variables which persist on the blockchain and can be altered.

Tokens Tokens can be described as a representation of an asset or a utility and are commonly also classified as a cryptocurrency. The endless possibilities of a smart contract application platform allows it to implement virtually any financial structure on it. Tokens are a separate currency which solely exists on the smart contract platform that it is programmed on. Technically speaking, a token is a deployed smart contract which keeps a register of balances for all addresses in its storage. With this register it keeps track of who owns how many tokens. The big advantage of this is the possibility to create your own cryptocurrency without a separate blockchain. All functions of tokens are run on the native blockchain of their platform which means they are also mined/-confirmed on the said blockchain.

The purposes of tokens are manifold: They can also differ significantly in their properties e.g., some have a limited supply while others are continuously minted. Most commonly tokens are used as a utility for a platform with blockchain functionality. However tokens can also be used to represent a security. For example, Maker developed a token which is pegged to the US Dollar's value via smart contracts [56].

Most tokens on the Ethereum network nowadays follow the ERC20-pattern [68] which is a token-standard. The standard describes necessary base functionalities of a token, e.g., transfer functions. Because most tokens follow this pattern, many products, like wallets or investment funds, only need to implement support for a single interface to be able to support the majority of all tokens.

In Subsection 3.2 we discuss how to use a token to represent a share in an investment fund and in Section 6.2 we cover the implementation used in the investment fund.

2.3 Blockchain related attack vectors

All cryptocurrencies combined handle transactions worth billions of USD every day [29]. This is a huge enticement for malicious parties to find a vulnerability in any of the blockchain's protocols or in smart contracts. Furthermore because of the anonymity of the blockchain it is possible to steal large sums without facing legal repercussions.

It is important to differentiate between vulnerabilities in the blockchain's implementation itself and vulnerabilities in smart contracts. Most famous bugs, such as theDAO attack [67], which lead to a direct loss of currency are actually bugs within the code of smart contracts and are not bugs of the blockchain. However, blockchains have some inherent, unavoidable security risks which we will also discuss.

Blockchain vulnerabilities Nearly all major cryptocurrencies rely on decentralization to ensure that no party is able to execute malicious transactions. A simple way to make use of publishing a malicious block is by *double spending* which describes the action of including transactions in a block spending more money than actually available. This is usually done by including two or more separate transactions spending all money available on an address. Afterwards, by confirming both of them, new currency is created (under the assumption that the block keeps being accepted). The attack can be sub-classified into other attacks such as the race attack or finney attack [8]. Double spending is hard to execute in reality, because it needs a significant voting/mining power of the blockchain. It is absolutely possible to publish a malicious block every now and then even with little mining power through the lottery system. For this very reason it is advised for anyone accepting cryptocurrency payments to wait for a safe amount of block confirmations. A block confirmation is any new block published after the block containing the transaction. If an attacker possesses 10% of all voting power the odds of getting three confirmations on their block is only 0.1%. It is up to the individual user to decide the amount of confirmations until accepting a transaction, for small amounts numbers around 6 for Bitcoin and 12 for Ethereum are commonly used. Exchanges often use a higher amount of confirmations because of handling large sums. The number of needed confirmations is also explored in the Bitcoin whitepaper [58].
By controlling significant voting power it is possible to conduct the commonly known *51% attack* which describes controlling the majority of the blockchains voting power and consequently taking control of the longest chain. Many people argue that the 51% attack is not a serious threat to any mature blockchain because by *taking over* the blockchain any value on it is simultaneously destroyed. If the trust system of a ledger is broken by centralizing it, it is easily detectable and would lead to parties stopping to accept the currency because of its unreliability.
Besides this well-known vulnerability, blockchains have to be resistant to many other attacks, such as the Sybil Attack [45] and the Denial of Service Attack [54]. Blockchains use different mitigation tactics for these problems. An extensive list of Ethereum's inherent problems can be found here [10].

Apart from these mostly unavoidable risks, the implementation of a blockchain is prone to bugs, just like any other software based product. For example Bitcoin introduced a large scale bug in 2013 [33] which caused nodes of newer versions to confirm blocks which where incompatible with older nodes. Consequently a split in chains was inevitable. This had brought damages to miners and enabled at least one large double spending attack. A split or fork in the area of cryptocurrency is a phenomenon where the main chain is split into two separate chains, either intentionally or by accident.
The Ethereum network suffered an arguably more severe bug in 2016; the Geth Consensus bug [39]. Similarly, by introducing a faulty update to the Go implementation of Ethereum a network fork was caused. A fork with 165 block containing thousands of transactions had to be abandoned and the valid chain was manually repaired. A hard fork is stronger than a split because it enforces a new rule set for blocks which makes the previously used one invalid. A soft fork introduces a new rule set which is backward

compatible. Bugs like these carry huge indirect financial damage to the blockchain, as they are a sign for instability and uncertainty.

Smart contract vulnerabilities and solutions Smart contracts behave similarly to any other program and as such are vulnerable to too many already commonly known risks. For example, one of these is the risk of an integer overflow and underflow, consequently many modern and secure smart contracts (especially tokens) use an integer extension which throws on an overflow or underflow [24].
Besides these there are vulnerabilities unique to smart contracts, in this section we will give examples of common, still active risks for Ethereum-based smart contracts.

Race conditions and Reentrancy Race conditions in smart contracts can occur when a function gives over control to another, often unknown, smart contract. At first it might not seem like this is something that can happen accidentally, but in Ethereum whenever a simple value transfer is executed, the default function of a smart contract is called. Basically any smart contract sending value to another unknown smart contract can potentially be vulnerable to race conditions. The default function can, when executed, call the original smart contract again which can lead to unexpected behavior.

```
1 // Vulnerable to reentrancy. Do NOT reuse.
2 mapping (address => uint256) public balances;
3
4 function closeAccount() public {
5     uint256 remainingBalance = balances[msg.sender];
6     require(msg.sender.call.value(remainingBalance)());
7     balances[msg.sender] = 0;
8 }
```

Fig. 5: Example of a smart contract vulnerable to reentrancy.

In Figure 5 an example for code allowing reentrancy is given. In line 6 the smart contract transfers the remaining balance to the requestor and consequently loses control of the program flow. The caller can reenter the `closeAccount` method and withdraw his balance multiple times. This is possible because the balance of the caller only gets reduced to zero after the external call is complete. So for every reentrance the balance is still in its initial state and allows for withdrawal.
Known solutions to any race conditions in traditional concurrent programming are locks or mutexes which can prohibit reentrancy. However race conditions are not limited to only reentrancy of the same function, but can also appear if the two functions use the same contract variable. It can be risky to use mutexes if the smart contract functions exhibit complex dependencies. The most commonly used solution is to avoid any state changes after external calls and restrict the gas usage of subcalls as much as possible. In our example above the reentrancy risk is solved if we switch line 6 and 7. It would

also be fixed if we would restrict the gas available to sub-call so that not enough gas for any other transactions is available. If it is not possible to do all state changes before an external call, a common pattern is to split functions into multiple others. This is known as the Withdraw-pattern or Pull-over-Push-pattern [27].

Front running Front running describes the attack of first observing a transaction of someone and then quickly starting a transaction and pushing it to execute before the observed transaction. Because transactions only get executed as part of a block in certain intervals, a few seconds or even minutes might pass before a transaction is included in the ledger. However after publishing the transaction it is visible to other nodes which can potentially extract information from it. Miners typically execute transactions which offer a higher reward first, therefore one can front run a transaction by starting a transaction with a higher reward. An attack like this is highly relevant to any market structures on the blockchain. For example, it is possible to observe someone planning to buy a lot of tokens from a decentralized exchange. With this knowledge someone could front run the transaction to buy tokens first which are then guaranteed to increase in value.

There is no easy fix for an attack like this, because it uses the inherent characteristics of blockchains. Market systems on the blockchain need to be designed in a way to not be affected by the order of transactions within a block. One possible design proposal for this would be to treat all transactions in one block equally regardless of their ordering (similarly to batch processing) by e.g., only making price updates take effect the block after they were mined.

Past smart contract attacks Attacks on smart contracts on the Ethereum blockchain have caused a direct damage of millions of USD to a large amount of blockchain users. This counts only the value of Ether and tokens directly lost through attacks. It is important not to overlook the indirect damages of any big vulnerability. When big companies fail to produce safe smart contract code it discourages blockchain investors to put money in blockchain technology by creating the fear of uncertainty.

One of the earliest accidents was the *theDAO* (decentralized autonomous organization) bug [67], which resulted in around USD 60M stolen. It was a well-orchestrated attack using multiple code security flaws with the main vulnerability being reentrancy. The attack was able to drain funds of theDAO's contract into a child DAO by executing code fragments hundreds of times more often than planned via a reentrancy vulnerability. Back in 2016 theDAO was the biggest project created on the Ethereum blockchain and held around 11 million Ether of around 80 million Ether total which means more than 10% of all Ether was in their smart contracts. Because of this large amount of funds theDAO was too big to fail and ultimately triggered a controversial hard fork to restore the funds lost in the attack [38].

Smart contract bugs are not a sign of inexperience: The company Parity Technologies has run into two successive, serious bugs related to one of their products, a smart-contract-based wallet. The company is led by the co-founder of Ethereum, Gavin Wood, and it is rumored that a large part of the multi-signature wallet code was written by him. The nature of the first bug is described well as a metaphor by BlockCAT: ”Imagine if you could walk into a bank and request that they transfer ownership of a stranger's account into your name. The bank happily does so, and you withdraw all of the account's money" [22]. The bug was really as simple as it sounds: Parity forgot to mark the `initWallet` function, which sets ownership, as `internal` (equivalent to private functions in other programming languages), see Figure 6. This bug lost around 150,000 Ether (around USD 30M at the time of occurrence), but all together around 0.5M worth of Ether were affected. The rest a group of white hat hackers retrieved and returned after a bug fix.

```
function initWallet(address[] _owners, uint _required, uint _daylimit) {
  initDaylimit(_daylimit);
  initMultiowned(_owners, _required);
}
```

Fig. 6: Vulnerable code in an earlier version of Parity's multi-sig wallet [16].

The second bug was again related to their multi-signature wallet (it even affected the same method: In a later version their wallet code relied on an external library which helps to reduce the creation costs of new wallets. Libraries usually do not fulfill any purpose by themselves but provide code functionality to other contracts which implement them. Nevertheless, libraries are still technically smart contracts, specifically, this library implemented all functions of the multi-signature wallet, including the initWallet function. Nobody had called this function on the library before until unknown user *devops199* took ownership and then called the `kill` function. They commented on Parity's Github page about their deletion with "anyone can kill your contract. I accidentally killed it." [44]. By taking ownership it was not possible to steal any funds, because the library itself does not handle them, however by deleting the library all wallets using it, lost all their functionality. This means all funds of these wallets are effectively frozen forever and irretrievable as their function to transfer currency does not exist anymore. With this bug around 0.5M Ether (around USD 150M at the time of occurrence) were lost, mainly effecting the funding of some large token startups.
Bugs like these make it clear that formal verification is direly needed in the blockchain world, where a small mistake can lose millions of USD.

3 Designing an investment fund for the blockchain

In this thesis we use the term *investment fund* rather broadly; the term has a long history and has shifted in meaning throughout the last century. An investment fund pools capital from a range of different investors, which can be wealthy individuals or professional investors. This capital can be used for many different means of investment, however this implementation is mainly focused on the assumption that the capital is actively managed to potentially offer more returns and reduce risk.

Classical investment funds are often refined into other types of funds such as a mutual fund, an exchange-traded fund (ETF) or a hedge fund. In the sector of cryptocurrency these more detailed descriptions can only be applied in a limited sense, because they are bound to legal differences and traditional investment goods. For example a cryptocurrency fund could arguably be characterized as all three types, because it is investing in currency only (mutual fund), it is publicly tradable (ETF) and it can aggressively invest and try to hedge risks (hedge fund).

The blockchain offers the technology to provide all functionalities that a fund typically uses. People can send currency to a smart contract which then pools the investment and can use it from a central point. This smart contract can hold all different ERC20-compatible tokens by design without needing special adaptations (introduced in Section 2.2). Parts of the managed funds can be easily accessible for active trading while others can be stored inaccessible in a cold wallet (introduced in Section 3.5). An investment fund can have shares by creating its own token via a smart contract supporting price management. These shares/tokens can be actively traded between participants and, if enough money is managed, can hit a public exchange. New shares can be dynamically created and burned depending on market need and based on the current price of the assets under management (AUM). It is possible to collect all kinds of fees, such as investment, withdrawal and management fees.

Besides all these functionalities an investment fund based on the blockchain also offers new features; for example, it is not needed to trade shares via a traditional authority - they can be traded peer to peer. It is possible to provide safety for the assets under management without the need of a governing body as well as provide full transparency of all assets.

This section aims to cover all basic functions of an investment fund, seen from a technological perspective based on the Ethereum blockchain. It will first cover the expected behavior and offer approaches on how to implement aforementioned behaviors. By going into detail on the approaches, we aim to highlight the features but also limits of this kind of cryptocurrency fund. We will discuss possible additions to the basic structure which are not seen in a traditional investment fund. Furthermore we briefly describe possible implementations to lay a basis for the blockchain semantics introduced in Section 4.

3.1 Funding

Funding on the blockchain is a well-explored topic since the inception of Ethereum and the start of token launches. As we discussed in Section 2.2 tokens currencies based on

another smart contract application blockchain, like Ethereum. New tokens try to offer a novel application or promise financial gain to token holders. Similar to traditional business they often try to secure funding to create their product.
Nowadays it is the quasi-norm for new token-based businesses to gather funds with crowdfunding. Specifically, crowdfunding for the creation of a new token is usually held as an *initial coin offering* (ICO) where companies sell (future) tokens over a smart contract to use for their advertised purpose. ICOs are not exclusively used for tokens but can also be utilized when creating a new blockchain. Tokens usually do not represent a share in the company, but are simply to be used later on. However tokens can increase in value if the usage of tokens increases or the company appears to make significant progress in their development. ICOs are well known for raising exorbitant amounts of money which are completely incomparable to traditional investment sums. For example at least 50 ICOs in the past have raised more than USD 25M [4], several have raised more than USD 100M [15] [14] and messaging app giant Telegram has raised USD 1.7B in two private presales [19] [20] for their future cryptocurrency *Telegram Open Network*.
The fund conceptualized in this paper will use tokens as a medium for shares, which makes an ICO a possible initial funding method. Often newly created tokens have a fixed token supply which will not change in the future. By offering a fixed supply in an ICO for an investment fund it would classify the fund as a *closed-end* fund, meaning it cannot take any further investments.
Another funding option is the continuous sale of new tokens. This can be achieved by augmenting a traditional token smart contract with *minting* features. The term minting is taken from traditional currency and simply describes the creation of new tokens. With a mintable token it is possible to create an *open-ended* investment fund, which means the fund can take on new investments at any time and is not limited to the fixed amount raised with crowdfunding.
Specifically for an investment fund we found the funding method of continuously selling new tokens more versatile and it is a novel approach which is not yet used by many companies. Additionally it would not prove difficult to transform an open-ended fund into a closed-end fund, while the other way around would need technical adaptions.

3.2 Tokens as shares

Tokens are a highly customizable currency and it is possible to adapt them to represent shares in companies or investment pools. For tokens to represent shares in a fund they should fulfill several purposes:

1. Tokens need to be directly bound to the investment fund.
2. Tokens represent a fraction of the assets of an investment fund which binds them to the net asset value.
3. Tokens need to be tradable between holders and potentially exchanges.
4. Tokens should be able to be created if new investors want to join the fund.
5. Tokens should be able to be liquidated to a commonly used currency at any time.

The above mentioned requirements will be addressed in this section and can help to give an idea on how the architecture of an investment fund based on smart contracts can be designed.

1. Token connection In order to satisfy point 1 it is clear that the tokens need to be directly connected to the investment fund contract. It is state-of-the-art to implement tokens as a smart contract which extends from commonly used patterns, in order to fulfill the ERC20 pattern [68] introduced in Section 2.2. Tokens fulfilling the ERC20 standard can be used by wallets, which offer support for the ERC20 pattern, without any special adaptions needed. This would leave the choice of implementing fund structures directly on a token or to statically bind it in an additional fund smart contract. The first option has the upside of reducing the overall amount of code needed and potentially reduces attack vectors related to the final product. The second approach falls in line with the commonly known *separation of concerns* software pattern and would reduce code complexity while increasing extendibility through modularity.

2. Token value Tokens serving as a part of the net asset value (NAV) of an investment fund are a tricky problem for multiple reasons. The token value needs to be represented as a single currency while the net assets consist of different currencies of which the exchange rate constantly fluctuates. If new shares should be mintable and old shares should be liquidatable the total assets can change at any given point. Furthermore for fund managers to manage assets they need to use external exchanges or smart contracts. While assets remain on exchanges and leave the fund's smart contract, they can be impossible to track.

These requirements make it clear that the price of a token cannot accurately be stored in a smart contract because it would need to be updated in an interval of seconds. An option to always retrieve the correct price would be to contact a traditional web-service through an oracle service such as Oraclize [25] on every purchase or liquidation. Contacting external resources brings many security risks because both the web-service and the oracle service could potentially turn malicious. Because cryptocurrencies are not observable if they are transferred to a centralized exchange, it would restrict the usage of these exchanges and require managers to only trade assets over suitable decentralized exchanges or atomic swaps [30] and create an enormous overhead of correctly tracking funds during an exchange.

All in all, price discovery on the blockchain is an immensely complex problem which would arguably be more time-consuming and difficult to implement than all other parts of an investment fund combined. For this reason we decided that we assume that an external, trusted party supplies a correct price to the fund's smart contract because price discovery would by far exceed the scope of this work.

3. Token trading In Section 2.2 the ERC20 pattern was mentioned as a standardized interface which makes tokens easy to be used by any existing product such as wallets. For the token used in the fund it is not necessary to alter the way it can be transferred between different participants of the blockchain. It suffices to simply use one of the default implementations used by hundreds of existing tokens [23]. A standard ERC20 implementation provides holders of tokens with an already well-established functionality of trading.

4. Dynamic token creation If a token does not have a fixed supply and the creation of new tokens is not bound to mining, the token is commonly called *mintable* token.

Usually mintable tokens are a rare sight because there is no need for creating new tokens for most use cases (tokens can be split into tiny fractions if necessary). In the case of an investment fund a mintable token is useful to take on new investments without affecting the underlying NAV of a share. For a mintable token to be trustworthy the minting process has to be bound to a specific function in a smart contract. If this would not be the case the owner of the token could create as much tokens for themselves as they please.
In our case this would mean that new tokens can only be generated with a smart contract function which is only executable when an external investor wants to purchase shares of the fund. Based on the amount the investor sends to fund and the current price of a token new tokens are minted so that the total supply increases while the price of a single token stays unaffected.

5. Token liquidation In order to liquidate a token into a commonly used currency like Ether the *burning* of tokens is required. Burning describes the process of destroying tokens so that they can never be recovered. There are two ways to achieve this; by either sending tokens to the zero address (which cannot be accessed by design) or by implementing a burning function within the token's smart contract. Both of these methods achieve the same result and there is no benefit or disadvantage for either method. Burning tokens is a special case for the investment fund because the burning happens from a centralized smart contract (the fund). A holder of tokens can request the liquidation of their shares. For the burning process the fund's smart contract has to hold enough Ether to match the price of the shares to be sold. This can prove to be an issue in times of volatility because not all assets are available in Ether, consequently it is not guaranteed for a payout to be successful. If the payout can be made, the fund has to burn the right amount of tokens from the requestor and transfer the according sum in Ether.

Liquidity of shares - Prevention of a premium Closed-end investment funds can face several problems during management. Several closed-end investment funds have found success in the past, with the TaaS.fund [41] being one of the first ones. Taking the TaaS.fund as an example, it does not only allow for new investments but also offers no direct way of liquidation. In order for a share-holder to sell their shares they have to find a buyer themselves. This brings a disadvantage for any investor because it takes away part of the motivation for fund managers to perform well as they do not have to worry about investors withdrawing their money. The TaaS.fund only offers a profit-payout quarterly for any token holders and raised money cannot leave the investment fund.
Another danger of closed-end investment funds, even if they allow liquidation, is the emergence of a premium. A premium describes the difference between the market price of a share of an investment fund and the underlying NAV of a share. Particularly if a fund is in high demand a premium is a common sight. Most cryptocurrency funds have had a high premium in 2017, sometimes as high as 80%-100%, because the supply was far lower than the demand. Even today still, some funds sell at a high premium such as Grayscales's Bitcoin Investment Trust which has a premium of around 50% [13]. A premium does not negatively affect a share-holder as long as the demand does not

decrease, however if the demand for shares decreases it can fall all the way down to the NAV of a share which can create heavy losses and poses a huge risk for any investors.
Open-end funds solve the problem of a premium by creating new shares if the demand is high. Nevertheless open-end funds consequently face a different problem: Creating and liquidating fund shares decreases the liquidity of fund shares themselves. If it is always possible to create and liquidate shares at the perfect price there is no need for share-holders to trade shares with potential new investors instead of liquidating them. It is disadvantageous for the investment fund if investors directly cash out for the obvious reason that it decreases their total assets and additionally it requires higher amounts of cash directly available for anyone to liquidate their shares. Directly available cash (Ether in this case) cannot be actively managed and might provide less profit. For these reasons it can make sense to make the price of creating a new share slightly higher than the NAV of a share and decreasing the price for liquidation.

```
function transfer(address _to, uint256 _value) public {
    require(_value <= balances[msg.sender]);

    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(_value);
    emit Transfer(msg.sender, _to, _value);
}
```

Fig. 7: Simple version of a `transfer()` function of an ERC20 token.

3.3 Managing multiple currencies

An investment fund would hardly deserve its name if it was only limited to a single cryptocurrency. In this work we focus specifically on an investment fund built entirely on the Ethereum blockchain in the form of smart contracts. This brings some limitations to the currencies which are directly manageable. Many large cryptocurrencies run on their own blockchain and cannot be directly held on another blockchain (e.g., Bitcoin, Ripple, Litecoin). Holding these currencies would require an investment fund to manage wallets on a variety of different blockchains which can potentially not interact with each other.
In this work we focus on an investment fund which can hold Ether and all ERC20 tokens based on the Ethereum blockchain. This poses some limitations but nevertheless covers a large amount of cryptocurrencies, for reference, currently 44 out of the 100 most capitalized (by market cap) cryptocurrencies are based on Ethereum [12] [7]. ERC20 tokens, as mentioned in Section 2.2 offer a standardized interface, which makes it possible to implement functions in the investment fund suitable for all tokens at once. With that it is relatively easy to manage multiple different currencies in a single wallet because all ERC20 tokens implement e.g., a transfer function which should look similar to the one in Figure 7. The function first checks if the sender has enough tokens on their

balance in line 2, only if this condition is upheld the execution will continue. In line 4 and the next one the amount is first subtracted from the sender and then added to the receiver. Lastly, in line 6, the function emits an event which covers all the details of this transfer. Events on the Ethereum blockchain are publicly sent out and any listeners to this contract can record the transfer (similar to a standard observer pattern).

3.4 Possible fee structures

For an investment fund to actively manage cryptocurrencies and try to achieve profit, it needs a method of earning money by doing so. Nowadays we are used to a number of different fee structures from traditional investment and hedge funds. A few fee possibilities are introduced and we discuss how they can be implemented on the blockchain.

A simple and commonly used fee is a fee paid upfront upon investment. Especially to our knowledge all token-based, closed-end investment funds currently on the market have made use of an upfront fee during their ICO. Because of the scarcity of cryptocurrency funds these fees are significantly higher than in traditional sectors. For reference one of the biggest closed-end funds Crypto20 currently manages around USD 75 million worth of assets [6] and is providing an index fund[1] of the 20 biggest cryptocurrencies. During its ICO it collected a fee upon investment of around 14.75% [61], similarly the Taas.fund collected a fee of 15% [41]. Both funds had big success in their funding round which proves even a high fee can work in a niche market. A fee like this can be easily implemented in an open-end fund by simply collecting it upon new investments by minting less tokens. Unlike closed-end funds an open-end fund also offers a liquidation option which would make it theoretically possible to collect fees if share-holders want to liquidate their shares. Fees collected upon the selling of shares are sometimes argued to be shady because funds should incentivize their share-holders to hold shares based on the fund's performance and no other factors. Utilizing it should be done with caution.

Besides one-time fees it can be beneficial to collect on-going fees supporting the costs of daily operations. A universally used fee is the *management fee* which gets deducted from all investments at regular intervals. Traditionally a management fee is collected per annum and ranges from 0.5% to 4% of the assets under management (AUM) depending on the management structure (actively versus passively managed funds). Implementing a fee that is deducted yearly is tricky in a smart contract because that would mean that the date of investment would have to be tracked if the management fee is deducted for the first time one year after investment. Furthermore doing so would make trading tokens difficult as tokens which are close to their *fee collection date* would be worth less than others. Generally speaking, creating unequal tokens is unheard of, creates many technical challenges and is confusing for anyone trading these tokens. Therefore it is important to find another solution on how to collect management fees which treats all tokens equally but does not lead to a mass-selling on the day before fees are collected. A solution to this would be that the management fee is collected in

[1] An index fund keeps a fixed ratio of shares of all its tracked components.

tiny intervals, as small as daily, by simply removing a certain fraction of a percentage from the AUM. For example to convert a yearly management fee of 2% you could collect a 0.0055% fee every day. This treats all tokens equally and does not create any dates were selling or buying shares would be particularly profitable.

Lastly, hedge funds often use a *performance fee* which is a fee collected based on the profit the investment fund made during a time period. Performance fees are usually much higher than management fees because they only take a part of the profits, not the total AUM. Hedge funds often collect a management fee of 2% and a performance fee of 20%, in fact this structure is so common that it got the name *Two and Twenty* [47]. Again it is difficult to implement a performance fee if tokens can be created and liquidated at any time, additionally it is not possible to split a performance fee into small intervals because it can increase the amount of the fee drastically. A possible solution could be to track on-going profit for the current period and deduct the profit fee partly if the shares are liquidated before the performance fee collection date. Currently the investment fund presented later in this paper does not implement a function for collecting performance fees, but it is an option worth exploring in the future.

3.5 Security features

Investment funds built and managed entirely on the blockchain have a more difficult playing field regarding trust and transparency issues. Traditional funds face strict laws regarding operation and distribution of shares. An investor does not have to worry about their money getting stolen or used for illicit activities. Crowdfunding and investing in new tokens or cryptocurrencies is often not monitored by governments and transactions in cryptocurrencies are practically impossible to track down if done correctly. For reference, just recently a Vietnamese company launched an ICO for the Pincoin which raised approximately USD 660 million. The team behind the ICO then went missing and the funds raised along with them [35]. Countless of other scams have happened in the cryptocurrency world and there is little to nothing anyone can do to recover the stolen money until a legal framework is established.
Many people who are experienced in the cryptocurrency investment field have a healthy suspicion towards any new ICO on the market, therefore the investment fund will introduce features aimed at increasing transparency and reducing the fear over an investment.

Multi-signature protection An investment fund can potentially manage large amounts of currency. In this case it is risky to have a single address/account control the whole fund. It is prevalent to protect smart contracts which hold large sums with a multi-signature contract. This means that the smart contract has a defined set of owners and a defined number of required approvals for every function it offers. For example, a multi-signature wallet can have ten owners and require at least five approvals for every transaction. Approvals are usually provided by sending a transaction to the smart contract from one of the owner's addresses (most known implementation of such a wallet is the Gnosis MultiSigWallet [9]). Alternatively it is also possible to sign the transaction offline and provide all signatures in a single transaction.

Cold-wallet support A cold wallet describes an account or smart contract which stores currency and is only accessible by an account of which the private key was never in contact with any device with network functionality. For example, this could mean the private key is stored on an old laptop or on a piece of paper. The idea behind a cold-wallet is that it provides the ultimate security, if the private key never comes in contact with the internet it is not vulnerable to any digital attacks. Most cryptocurrency exchanges make use of cold wallets to protect any funds which are not needed to be in constant circulation. For reference, one of the biggest cryptocurrency exchanges Coinbase claims to keep 98% of customer funds offline [26].

Circle of trust If an investment fund is actively managed by fund managers it is necessary to move and trade currency at any time in order to react to the market. However most daily transactions should be between the same partner wallets: Moving funds from cold wallets into circulation and vice-versa and sending funds to cryptocurrency exchanges. Because most used destinations are known in advance the investment fund provides the feature to include addresses into a circle of trust. For all trusted addresses it is possible to require fewer approvals from the multi-signature contract to take an action. For untrusted addresses the fund can add so called trust-party owners which need to give their approval in addition to the internal management accounts. Trust-party accounts could be given to large investors in the fund, to exchanges or to other trusted persons. The reasoning behind untrusted wallets is that it is impossible for the fund managers and fund owners to quickly steal all currency by moving it to an unknown wallet.

4 Blockchain semantics

In this section we introduce a blockchain semantics with the goal of being able to describe smart contract calls and smart contract interactions. The semantics will be abstracted to a level representing what smart contract programmers can observe as opposed to what happens in detail during the execution of transactions on the Ethereum Virtual Machine.

The blockchain can be modelled as an automaton where the current state represents the balance, storage, etc. of all participants. Then transactions are used as state transitions to change the global state. We define transactions slightly different than how they are defined in Bitcoin's and Ethereum's white papers. Specifically, we treat external transactions and internal transactions (which are started by smart contract executions) equally. This will be covered in detail in Section 4.3, were we also introduce a big-step semantics which is capable of modelling complex smart contract executions possibly containing multiple internal calls.

The semantics in this section was created with a practical intention: It is possible to specify the functionalities of the investment fund presented in Section 6. In order to show the link to practice we define one concept used in the investment fund, the multi-signature wallets, and prove that their feature of trust circles, described in Section 3.5, is upheld.

4.1 Design choices

We introduce an abstract blockchain structure which represents the features of the Ethereum smart contract platform. We define three different concepts, namely contracts, the global state and transactions. Contracts are immutable entities of the blockchain and can be compared to the stored code of a deployed smart contract. The global state acts as a snapshot of the blockchain and saves all mutable information, such as account balances and contract variable values. Finally transactions are used in a big-step mechanism to step from one global state to another which closely resembles a real-world transaction.

Part of the definitions in this section were introduced first by Grishchenko et al. [48] who defined a semantic framework for the security analysis of Ethereum smart contracts. Their semantics represents the inner workings of Ethereum smart contract executions on a byte-code level. In this work we try to abstract common features, such as the storage of an account, and instead present contract state variables. Simply said, this work's semantic framework tries to resemble the structure of a user-centric programming language, such as Solidity, instead of tying itself too closely to the EVM bytecode, while still respecting the overall structure of the underlying computational model.

Tail Calls By introducing a big-step semantics that allows for an execution of a succession of transaction, we chose to make an important limitation for how smart contracts can start other transactions. Normally it is possible to start an arbitrary number of new transactions within a single smart contract function. However our semantics only allows for tail-calls. This means that we assume a transaction can only start another transaction

as its very last action.
This design choice was made because it drastically reduces the complexity of the big-step semantics. Additionally for all functionalities of an investment fund tail-calls will be sufficient and many other smart contract functions can be modelled to only have tail-calls by moving subsequent code to the next function. In practice smart contract functions often execute calls as a last action in order to eliminate the risk for reentrancy (covered in Subsection 2.3).

Nested Errors If a function is normally called on the Ethereum network it provides information about the successful execution and returns a boolean value representing the success of the call. The creator of a smart contract can choose to act based on this information. However there are only rare use cases where it makes sense to ignore the execution status of a call. For all our functionalities of the investment fund we will require successful execution to ensure the expected outcomes. In Solidity this is simple to achieve by surrounding outgoing calls with *require*$\{\dots\}$. We will define that if a transaction is followed by another transaction which results in the error state, the original transaction will result in the error state as well.

Error-proneness We purposely leave out one of Ethereum's core properties: Gas costs. Normally every transaction on the blockchain has a gas limit specified by the initiator of the transaction. Gas is used to pay to miners who execute the transaction. Every operation in a smart contract has a gas cost and consequently if one runs out of gas, the transaction woill fail and all gas will be consumed. Because of this characteristic of the blockchain it is important to take into account that every transaction (and also every nested transaction/call) can fail if not enough gas is given to it. It is crucial that this is reflected in our abstraction in order to ensure that it is never possible to guarantee that a transaction will succeed. In order to model this behavior in our semantics without delving into introducing gas directly, we abstract this behavior by specifying every function can always potentially fail. Including gas cost in our semantics would be infeasible as we do not characterize functions in an operational but rather in a declarative fashion due to the high abstraction level. Even though the resulting semantics is non-deterministic, while real smart contracts are deterministic, we believe that this is no restriction for reasoning about contracts.

Public contract states Every deployed contract on the Ethereum blockchain has a contract state or storage associated with it. Utilizing the state, the contract can keep a persistent memory throughout multiple transactions. Normally variables saved in the contract state are not accessible directly by other contracts, unless a getter-method is included in the contract code. In Solidity contract variables can be annotated with `public` in order to automatically create a getter-method on deployment. Even though the contract state can be made private and inaccessible for other contracts, it is by no means private for observers of the blockchain. Every change to any contract state can be observed in the transaction included in the blockchain.
Our semantics assumes that all contract variables saved in the contract state are publicly accessible for other smart contracts. Additionally, there will be no sub-transaction

to specify the action of reading another contracts state. The transaction is simply able to access the information without any action. Even though this behavior is not in line with the actual behavior of the blockchain, we believe it does not restrict any functionalities or makes infeasible assumptions as it is rare to see private contract variables in contracts nowadays. It would be possible to introduce private variables by accessing every contract's state via getter methods. This would make the semantics significantly more verbose which is the main reason for this abstraction.

Fallback functions Solidity allows for the use of fallback functions which get executed if a smart contract is called with a function signature that is not defined in the contract code or if no function signature is given (simple value transfer). Fallback functions were used for a large number of past smart contract related attacks (some of which are covered in Subsection 2.3) because it is possible to execute any code in the fallback function. In our semantics we do not explicitly support fallback functions, instead we define a `receive` function which is used to receive value from a value transfer and which does not have the functionality to execute any other code. There are trade-offs for omitting fallback functions, such as that no behavior can be modelled when value is received. However at the same time it reduces the complexity of the semantics and does not reduce expressiveness as long as interactions only happen between known smart contracts. If this semantics would be used to model interactions with unknown parties, it is necessary to add fallback functions to verify the outcome of a transaction.

External accounts and contract creation/destruction The semantics introduced in this chapter is solely focused on providing the tools to model smart contracts and their behavior. It currently does not include an explicit definition of external owned accounts which are basic accounts capable of holding Ether and initiating transactions but have no associated code. Nevertheless, on the Ethereum network only external accounts can be used to call a smart contract function (while this function can call other smart contract functions). For this reason we simply make the assumption that smart contract transactions are started, but do not specify how exactly this happens. This abstraction allows us to restrict the scope and cover exclusively the semantics for smart contracts.
Furthermore we do not cover smart contract creation/deployment and smart contract destruction which means that we assume that all smart contracts already exist and never cease to exist. Introducing smart contracts as a permanent structure does not significantly restrict the functionality of the semantics in our opinion. It is a common design pattern for smart contracts on the Ethereum blockchain to not implement a self-destruct function. Instead the contract is simply deprecated and left to exist while a new one is deployed at a different address. Moreover, self-destruction would also be easy to incorporate into the semantics if needed.

4.2 Preliminaries

Let $\mathbb{B}$ be the set of bits $\{0, 1\}$ and let $\mathbb{B}^k$ with $k \in \mathbb{N}$ denote the set of bitstrings of the size k. For bit- and byte-like types we make use of standard concatenation which we

write as $x^\frown y$ where $x, y \in \mathbb{B}^*$, in addition we assume implicit conversion to the little-endian representation from $\mathbb{N}$ to $\mathbb{B}^*$. In this work we define arrays with the notation $[X]$ and in order to more easily access elements of tuples and arrays we use a special access annotation: $P.p_1$ accesses the element p_1 of P and $P[1]$ accesses the first element of the array. Additionally we use the element notation $\in$ to state that an element is in an array. In order to update functions and tuples as part of inference rules, we make use two distinct update notations and abbreviations for sums and subtractions. To update an element of a function we write $f = f\langle x \rightarrow y\rangle$ which means that the function f is changed so that the element x now maps to the element y. To update an element of a tuple we write $T = T[x = x + y]$, which means that the value of element x is overwritten by the sum of $x + y$. An update like this can be abbreviated with $T = T[x \mathrel{+}= y]$, subtractions can be abbreviated analogously. The aforementioned update notations can also be chained, like so $f\langle x \rightarrow y\rangle\langle a \rightarrow b\rangle$, in order to update multiple elements at one.
For inference rules we make use of the conditional operator ? in a standard way.

Types For our theoretical implementation of the Ethereum blockchain we define a set of static types. In Section 4.3 we will introduce the global state of the blockchain for smart contracts. Every smart contract has an array of contract variables which is immutable in length and furthermore each element is immutable in its type.
Theoretically values of every type which is currently usable by Solidity can be represented by a bitstring of a certain length. Keeping in mind usability and clarity we define natural numbers in addition to bitstrings as primitive types. We also define the commonly used structures of arrays and mappings, these definitions are recursive with the exception that a mapping must use a primitive type as a key (matches the current Solidity implementation).
Solidity also allows for dynamic structures called *structs* which are also common among other programming languages and make it possible to define new types out of a list of other types. For all use cases we present in this paper we do not require structs and therefore leave them out from the type definitions.
Formally we represent all needed types as follows:

$$\begin{aligned} x \in \mathbb{N} : TPrims \ni \mathit{tprim} &:= \mathit{bitstring}^x \mid \mathit{natural} \\ Types \ni \mathit{type} &:= \mathit{Array}(\mathit{type}) \mid \mathit{tprim} \mapsto \mathit{type} \mid \mathit{tprim} \end{aligned} \tag{1}$$

We write $natural$ to reference the type of natural numbers $\mathbb{N}$ and $bitstring^x$ to reference the type of bitstrings $\mathbb{B}^x$. These two types make up the primitives set. In addition we then define the dynamic structures as $Array(type)$ and $tprim \mapsto type$ which correspond to array and map structures. Furthermore, let $typeof$ be a helper function which can take any value as an input and returns the type of the value.
For values we assume that the annotation determines their type, i.e. $n_{natural} \in \mathbb{N}$, $b_{bitstring^{160}} \in \mathbb{B}^{160}$. For complex types we assume that all elements contained within them also follow these rules. For example, if we write $A_{Array(natural)}$ it represents an array of natural numbers, so that all elements within this array are natural numbers.

Next we formally define the above described properties mutually recursive:

$$\begin{aligned} typeof : vals \to Types \\ \forall t, \forall s \in S_t : typeof(s) = t \end{aligned} \tag{2}$$

$$S_t := \{v \in vals \mid typeof(v) = t\} \tag{3}$$

$$\begin{aligned} vals := n_{natural} \mid b_{bitstring^x} \mid A_{Array(t)} \mid M_{t_k \mapsto t_v} \\ \text{with } n, x \in \mathbb{N},\ b \in \mathbb{B}^x,\ t, t_v \in Types,\ t_k \in TPrims,\ A \in [S_t],\ M \in S_{t_k} \to S_{t_v} \end{aligned} \tag{4}$$

Now that we specified the details elements of a certain type need to fulfill we can construct our total set of values needed. Let $\mathcal{V}$ be the set of all values $vals$ representable in the global state of the blockchain.
Additionally we define separate annotations for arrays and mappings. We call $\mathbb{A}_t := S_{Array(t)}$ the set of all arrays of type t and call $\mathbb{M}_{t_k \mapsto t_v} := S_{S_{t_k} \mapsto S_{t_v}}$ the set of all mappings mapping from t_k to t_v.

4.3 Semantics

In this subject we introduce all parts of the semantics formally and describe the big-step semantics. Because most parts depend on each other or are related we give a summary of them in Table 1 which can also act as a quick reference point.

Global state σ	The current state of the blockchain is represented by the global state. It acts as a snapshot of all information (balances, contract states and contracts) at a certain moment and maps addresses to contract states.
Contract C	Contracts reflect the immutable parts of the blockchain and correspond to the contract code of a contract deployed on a blockchain. They contain information about the address, contract variables (storage) and contract functions.
Contract state S	The contract state gives information about the current state of an address in the global state. It contains all mutable parts, namely the balance and the assignment of contract variables (storage) and links to the contract which is immutable.
Transaction T	Transactions contain information about the initiator and actor (receiver) of a transaction, the function executed by the transaction and its parameters as well as the value sent with the transaction. Transactions are used in the context of a transaction environment Γ to step from one global state to another $\Gamma \vDash \sigma \to \sigma'$.

Table 1: Summary of the basic components of the semantics introduced in this section.

Contracts In order to reflect the immutable parts of the blockchain we define a contract C as a tuple (a, S_V, S_F, F) where $a \in \mathbb{B}^{160}$ refers to the address of the contract, $S_V \in Names \times Types$ is a set of tuples consisting of variable signatures which are pairs of the variable name and the variable type where $Names$ is the set of all contract variable names. $S_F \subseteq \mathcal{F}$ is the set of signatures of supported functions by the contract. Lastly F is the functionality of the contract which maps function-signatures f to all possible outcomes when the function f is executed in a transaction. A function signature is a unique identifier for a function of a smart contract and we call the set of all function signatures $\mathcal{F}$. Furthermore we assume that all signatures can be implicitly be converted to bitstrings and can therefore also be saved in the state and passed as arguments.
Formally we denote

$$F \subseteq \mathcal{F} \to \mathcal{P}(\mathcal{T}_{env} \times \varSigma \times \mathbb{A}_{\mathcal{V}} \times (\mathcal{T}_{env} \cup \{\epsilon\} \cup \{\textsf{ERR}\}))$$

where $\mathcal{T}_{env}$ is the set of all transaction environments, $\varSigma$ the set of all global states, ϵ represents the end state and $\textsf{ERR}$ represents an error during execution of the transaction.
This means that the functionality maps function signatures to their behaviors which can be seen as an input-output relation. The input of a functionality can be seen as the first two elements out of the sets $\mathcal{T}_{env}, \varSigma$. The first element is the transaction environment $\varGamma$ which contains all necessary information about the transaction being executed and additionally meta information about the current block. The global state σ chosen of the set of global states $\varSigma$ is the initial state on which a transaction is executed on, so we are observing how the global state changes under the influence of a transaction environment $\varGamma \in \mathcal{T}_{env}$.
The latter two arguments $\mathbb{A}_{\mathcal{V}}, (\mathcal{T}_{env} \cup \{\epsilon\} \cup \{\textsf{ERR}\})$ represent the outcome of the aforementioned transaction environment being executed on the global state σ. The array set $A_{\mathcal{V}} \in \mathbb{A}_{\mathcal{V}}$ contains the new contract state variable assignment of the address of the functionality, because as part of the transaction relation introduced in Section 4.3 this array replaces the variable values as part of the execution and therefore represents the state change. The second output argument states whether another transaction is started by the functionality. The end state ϵ signalizes that the transaction ends. If another transaction environment is given the functionality has a sub-transaction and if the function error $\textsf{ERR}$ is given it signalizes that the function would end in an error. How exactly transactions and sub-transactions work is covered in Section 4.3.

Example Let us see how the contract tuple can be used to represent a real-life smart contract. We will continue to extend this example as we go on and introduce more features to our semantics.
Here we present a simple wallet smart contract: It should be able to store, send and receive Ether. Additionally we want the wallet to have an owner, so that it can only be used by one address. The contract consists of the tuple (a, S_V, S_F, F), a is an arbitrary but unique address e.g., `0x111a11e7...` The variable signatures in our case only contain the owner which is a variable of the type address $S_V = \{(\text{OWNER}, bitstring^{160})\}$, where the first element represents the name and the second the type. The supported functions are as we mentioned a receive and transfer function (the receive function can

also be seen as a payable default function in the Solidity sense)
$S_F = \{\text{RECEIVE}, \text{OTRANSFER}\}$. Finally the functionality F set contains all possible outcomes for every execution of a function (an immensely large set). We will see how the functionality can be inferred as we continue this example throughout this section.

Transaction environment The transaction environment represents the details of a transaction such as the sender, the value and block information. While we do not make use of block headers in this work, by defining them in the transaction environment we make sure that the semantic framework is easily extendable to other applications. We say a transaction environment $\mathit{\Gamma} \in \mathcal{T}_{env} = \mathbb{B}^{160} \times \mathcal{T} \times \mathcal{H}$ is a tuple of the form (o, T, H) where $o \in \mathbb{B}^{160}$ is the origin address of the transaction, $T \in \mathcal{T}$ is the transaction being executed and $H \in \mathcal{H}$ is a tuple of all relevant block information. Note that the origin address of the transaction is not always equivalent to the $init$ address of the transaction. Say if a transaction initiates a chain of other transaction in course of its execution, the origin address will stay constant throughout this chain, while the initiator changes based on the transaction. We do not define elements of $\mathcal{H}$ here, but it commonly contains information such as the block number and the block timestamp.

Global state The current state of the blockchain is represented by the global state $\sigma : \mathbb{B}^{160} \to \mathbb{S}$. Where $\mathbb{S}$ is the set of all contract states, a contract state $S \in \mathbb{S}$ is a tuple of the form $(b, V)_C$ with $b \in \mathbb{N}$ representing the balance in wei, the minimal fraction of the native currency Ether. $V \in Names \times \mathcal{V}$ represents the set of contract variable assignments which are tuples consisting of the variable signature and the value of the variable. C is the contract of the contract state, which is written as an annotation to signalize that it should be immutable.
The global state contains information about all participants of the blockchain. It can be imagined as a register which updates all balances and states after each transaction. Besides balances, the global state also contains the value of contract variables, the smart contracts' persistent memory. The contract variables can contain crucial information such as all balances for any token created on the blockchain. Balances and contract variables are subject to change if a transaction interacts with the global state, while contracts themselves and address associations are immutable.

Example cont. We continue with our wallet contract introduced in the example above. Because the contract only represents the actual immutable code of a smart contract the global state includes the balance of this wallet as well as the current owner address of the wallet. The current state is mapped to address $\sigma(\texttt{0x111a11e7...}) = (100, V)_C$, as an example we state here that the contract is holding a balance of 100 wei[2]. The second argument of the tuple is the current assignment of contract variables which only includes the owner $V = \{(\text{OWNER}, \texttt{0xdebb1e...})\}$

[2] wei is the smallest fraction of Ether with 1 Ether = 1,000,000,000,000,000,000 wei

Transactions as a big-step relation We call $T = \mathbb{B}^{160} \times \mathbb{B}^{160} \times \mathcal{F} \times \mathbb{A}_{\mathcal{V}} \times \mathbb{N}$ a transaction which is the tuple $(init, actor, f, par, val)$ where $init$ is the address initiating the transaction. The $actor$ is the address performing the execution and contains the smart contract code which will be executed. With f we refer to the signature of the functionality being executed by $actor$. The list of parameters par is supplied to f which can be used as input during the smart contract execution. The number val is the value of wei sent with the transaction and will affect both the balance of the initiator and actor.

In our semantics transactions are used to step from one global state to another. Broadly speaking these transactions resemble transactions on the Ethereum blockchain with the exception that a transaction can both be an *external* or *internal transaction*. Moreover we make a clear differentiation between transactions defined in this work and transactions on the Ethereum blockchain.
Ethereum allows for the execution of external transactions, these newly started transactions each have a separate entry on the blockchain and can call smart contract functions. A smart contract cannot initiate an external transaction, only an external account can. We do not specifically define external accounts in this work but simply assume that they exist and can start transactions (see Subsection 4.1).
However a smart contract deployed on the Ethereum blockchain can execute code of another smart contract by explicitly calling it with an internal transaction. These internal transaction do not have a separate entry on the blockchain (however some modern block explorers like etherscan.io [11] list them separately) but nevertheless can still transfer Ether. This means theoretically a hundred value transfers between a hundred different addresses could happen with there being only a single transaction logged on the blockchain. In Solidity an external transaction can be initiated with functions such as `call()` and `transfer()`.
As we could observe, both external and internal transactions can have a significant impact on the blockchain while only one type is explicitly logged. Because of this reason we made the design choice of defining both types of execution as transactions as it allows for easier analysis of the effects of executing a certain function in a smart contract. By doing so we do not limit the mightiness of the original blockchain, because we can still represent a complicated external transaction with many internal transactions as a single transaction which consists of several others.

Example cont. For our owned wallet contract introduced in our example so far a possible transaction would be

$$T = (\textsc{0x57a27...}, \textsc{0x111a11e7...}, \textsc{otransfer}, [\textsc{0x700000...}, 100], 0).$$

This transaction states that it is initiated by the $\textsc{0x57a27...}$ address and calls the $\textsc{otransfer}$ of our wallet contract. Moreover, it passes two arguments $\textsc{0x700000...}$ and $amount$ with the function and does not send value (last element is 0). We will see how this transaction is executed in the next example.

Big-step transaction semantics A transaction T can be used for a big-step execution on a global state σ which will execute a function f in the contract of the actor and can change states of the initiator and actor. In addition the function can optionally start another transaction or exit with an error. To describe the execution of a transaction we write $\varGamma \vDash \sigma \to \sigma'$. In the following we define possible outcomes of a transaction, first we define a transaction after which no other transaction follows:

BASESTEP

$$\frac{\begin{array}{c} C = (a, S_V, S_F, F) \qquad (\varGamma, \sigma, V', \epsilon) \in F(f) \\ \varGamma.T = (\mathit{init}, \mathit{actor}, f, \mathit{par}, \mathit{val}) \qquad \sigma(\mathit{actor}) = (b, V)_C \qquad \sigma(\mathit{init}).b \geq \mathit{val} \\ f \in S_F \qquad \sigma' = \sigma\big\langle \mathit{init} \to \sigma(\mathit{init})[b \mathrel{-}= \mathit{val}]\big\rangle\big\langle \mathit{actor} \to (b + \mathit{val}, V')_C\big\rangle \end{array}}{\varGamma \vDash \sigma \to \sigma'}$$

The rule above describes how a transaction T of the transaction environment $\varGamma$ influences the global state σ. We can observe that the balance specified by init is transferred to actor and that the transaction can only be successful if enough balance is on the contract of init and if the contract has an implementation of the function f to execute ($f \in S_F$). The functionality F specifies a new assignment for the contract state variables V' by supplying signature f like so $F(f)$. These new contract state variables replace the current ones of the contract actor in the new state σ'. The ϵ signalizes that no other transaction follows after this one.
In short, the balance is transferred from initiator to actor and the values of the contract variables of the actor can change based on the functionality of the contract.

Next we will show the effect of a transaction which throws an error. We define $\bot$ as the error state of the global state. It signalizes that the global state cannot be changed further by other transactions. While the Ethereum blockchain does not enter an error state by itself, it is possible for transactions to result in an error. Therefore we can compare the result of the global state stepping into an error state as a simple REVERT action, which means if we transition into an error state nothing happens to the global state.

FUNCTION ERROR

$$\frac{(\varGamma, \sigma, V', \mathsf{ERR}) \in F(f)}{\varGamma \vDash \sigma \to \bot}$$

As we can observe above, if a transaction throws an error the global state enters the error state. That means neither the value of the transaction is transferred nor the state of the actor is changed. This behavior matches the one of the Ethereum blockchain, with the exception that we do not consider gas. Normally gas is also burned on errors to pay for the execution of a functionality till the error occurs. In Section 4.1 we discuss the abstraction of excluding gas from our semantics.

In addition to entering the error state because of running out of gas, the error state could also be entered if the initiator has insufficient funds to send the specified value as the next rule shows.

$$\textsc{Insufficient Balance}\quad \frac{\mathit{\Gamma}.T = (\mathit{init}, \mathit{actor}, f, \mathit{par}, \mathit{val}) \qquad \sigma(\mathit{init}).b < \mathit{val}}{\mathit{\Gamma} \vDash \sigma \to \bot}$$

Next we define how a transaction behaves when the functionality starts another transaction. This chained transaction is the equivalent of an internal transaction on the Ethereum blockchain and can potentially start even more internal transactions itself.

$$\textsc{Recursive Step}\quad \frac{\begin{array}{c} C = (a, S_V, S_F, F) \qquad (\mathit{\Gamma}, \sigma, V', T') \in F(f) \\ \mathit{\Gamma}.T = (\mathit{init}, \mathit{actor}, f, \mathit{par}, \mathit{val}) \qquad \sigma(\mathit{actor}) = (b, V)_C \qquad \sigma(\mathit{init}).b \geq \mathit{val} \\ f \in S_F \qquad \sigma'' = \sigma\langle \mathit{init} \to \sigma(\mathit{init})[b \mathrel{-}= \mathit{val}]\rangle\langle \mathit{actor} \to (b + \mathit{val}, V')_C\rangle \\ \mathit{\Gamma}' = \mathit{\Gamma}[T \to T'] \qquad \mathit{\Gamma}' \vDash \sigma'' \to \sigma' \end{array}}{\mathit{\Gamma} \vDash \sigma \to \sigma'}$$

This inference rule gives us a definition of function calls on the Ethereum network. It is similar to a transaction without a sub-transaction call (signalized by ϵ), in the sense that it has the same effect on the global state by changing the state of the actor and the balance of the initiator and actor. However in addition there can be arbitrary many sub-transactions which are created as an effect of T and possibly T'. These transactions can also have an effect on the balance and state of their initiators and actors. By stating $\mathit{\Gamma}' \vDash \sigma'' \to \sigma'$ and defining the value of σ'' we allow freedom to how the next transaction influences the global state.

Example cont. In our example up to now we have introduced the contract of an owned wallet, an example contract state and a transaction. Furthermore we have specified that the wallet has two functions, a receive and a transfer function, but have not described these. Here we continue with the transaction introduced in the last example and state how its outcome can be inferred.

$$\textsc{otransfer}\quad \frac{\begin{array}{c} \mathit{to} \in \mathbb{B}^{160} \qquad \mathit{amount} \in \mathbb{N} \\ \mathit{\Gamma}.T = (\mathit{init}, \mathit{actor}, \textsc{otransfer}, [\mathit{to}, \mathit{amount}], 0) \qquad \sigma(\mathit{actor}) = (b, V)_C \\ (\textsc{owner}, v_{\textsc{owner}}) \in V \qquad \mathit{init} = v_{\textsc{owner}} \qquad T' = (\mathit{actor}, \mathit{to}, \textsc{receive}, [\,], \mathit{amount}) \end{array}}{(\mathit{\Gamma}, \sigma, V, T') \in F(\textsc{otransfer})}$$

As we can see by the definition of the OTRANSFER function, it transfers Ether to the *to* address by calling their RECEIVE function with the specified *amount*. The wallet additionally has a single owner which we specified in the contract variables before. With the statement $\mathit{init} = v_{\textsc{owner}}$ we restrict the function so that only the owner can be the initiator of successful transfer. This inference rule makes use of the Recursive Step by calling the receive function of the recipient of the transfer. It is also important to recognize that the initial transfer transaction does not carry any value, because it is only used to initiate the transfer from the smart contract.

Next we have to introduce an inference rule to define how recursive function calls behave if an error happens at any point during execution. If any of the sub-transaction of a recursive call end in the error state, it is impossible to fulfill the inference rule of RECURSIVE STEP. Nevertheless from a practical standpoint it makes sense that the parent transaction also results in an error state in that case.

ERROR PROPAGATION

$$\frac{\begin{array}{c} C = (a, S_V, S_F, F) \qquad (\mathit{\Gamma}, \sigma, V', T') \in F(f) \\ \mathit{\Gamma}.T = (\mathit{init}, \mathit{actor}, f, \mathit{par}, \mathit{val}) \qquad \sigma(\mathit{actor}) = (b, V)_C \qquad \sigma(\mathit{init}).b \geq \mathit{val} \\ f \in S_F \qquad \sigma'' = \sigma\langle \mathit{init} \rightarrow \sigma(\mathit{init})[b -= \mathit{val}]\rangle\langle \mathit{actor} \rightarrow (b + \mathit{val}, V')_C\rangle \\ \mathit{\Gamma}' = \mathit{\Gamma}[T \rightarrow T'] \qquad \mathit{\Gamma}' \vDash \sigma'' \rightarrow \bot \end{array}}{\mathit{\Gamma} \vDash \sigma \rightarrow \bot}$$

Well-formedness of contract states and global states We define properties here to establish commonly shared patterns of blockchains. Informally we say that a deployed contract will always stay at the address it was initially deployed at and that the contract variables have a fixed type and new values assigned to them must be of that type which ensures that account states are in accordance with their associated contracts.
We defined S_V in context of a contract so that once a contract exists, the amount, name and type of class variables cannot be altered. We define this formally as follows.

Definition 1 (Well-formedness of contract states). *Let $S = (b, V)_C$ be a contract state and let $C = (a, S_V, S_F, F)$ be its contract. We call S well-formed if it fulfills these criteria.*

$$|V| = |S_V| \tag{5}$$

$$\forall n, v : (n, v) \in V \Rightarrow (n, \mathit{typeof}(v)) \in S_V \tag{6}$$

Next we define well-formedness on contracts so that the functionality upholds the well-formedness of the contract state if executed by a transaction. Also we add another property to the well-formedness regarding the initiator address of sub-transactions. Functions have the ability to execute another transaction after their execution, for this action it is important that we impose the restriction that this new transaction has to be initiated by the current actor. Basically if a function is executed by a certain contract and another transaction is started, this transaction cannot be initiated by any other party. This is an intuitive feature and is present on all major blockchains to our knowledge. Lastly, we also define that contract variables must have a unique name within the contract. Below we define this formally.

Definition 2 (Well-formedness of contracts). *Let $C = (a, S_V, S_F, F)$ be a contract, we call C well-formed if the following conditions hold:*

$$\begin{aligned} \forall \mathit{\Gamma}, \sigma, V', f, n, v, \forall T' \in \mathcal{T} \cup \{\epsilon\} : (\mathit{\Gamma}, \sigma, V', T') = F(f) \Rightarrow \\ (|V'| = |\sigma(\mathit{\Gamma}.T.\mathit{actor}).C.S_V| \\ \wedge (n, v) \in V' \Rightarrow (n, \mathit{typeof}(v)) \in \sigma(\mathit{\Gamma}.T.\mathit{actor}).C.S_V) \end{aligned} \tag{7}$$

$$\forall \mathit{\Gamma}, \sigma, V', f, \forall T' \in \mathcal{T} : (\mathit{\Gamma}, \sigma, V', T') = F(f) \Rightarrow \mathit{\Gamma}.T.\mathit{actor} = T'.\mathit{init} \tag{8}$$

$$\forall S_1, S_2 \in S_V, S_1 = (n_1, t_1), S_2 = (n_2, t_2) : S_1.n_1 = S_2.n_2 \Rightarrow S_1 = S_2 \tag{9}$$

We define a simple property to uphold the integrity of the relation between global states and contracts. Here we state that the address linking to a contract must match the address found within this contract. Additionally the contract deployed at an address is immutable and therefore cannot be changed through any transaction. This slightly restricts the real-world behavior where it is possible to `selfdestruct` smart contracts if needed which we discuss in Subsection 4.1.

Definition 3 (Well-formedness of the global state). *We call a global state $\sigma \in \mathit{\Sigma}$ well-formed if every contract state $S \in \mathit{Ran}(\sigma)$ is well-formed and if all contracts $C \in S$ of all contracts states $S \in \mathit{Ran}(\sigma)$ are well-formed. Additionally the global state needs to fulfill:*

$$\forall a : \sigma(a).C = (a', S_V, S_F, F) \Rightarrow a = a' \tag{10}$$

$$\forall \mathit{\Gamma} : \mathit{\Gamma} \vDash \sigma \rightarrow \bot \;\vee\; (\mathit{\Gamma} \vDash \sigma \rightarrow \sigma' \Rightarrow \sigma(a).C = \sigma'(a).C) \tag{11}$$

Fallibility property for all functions As we mentioned in Section 4.1 we do not consider gas explicitly in our transaction environment, however it plays an important role in the behavior of a transaction execution. Meaning if not enough gas is provided a normally successful function can fail. For this reason we chose to model this property of fallibility as an abstraction by defining a step which allows every transaction to fail.

Definition 4 (Fallibility of contract functionalities). *Let $C = (a, S_V, S_F, F)$ be a contract, we call the contract fallible if it fulfills the following property.*

$$\forall f, \mathit{\Gamma}, \sigma, V : (\mathit{\Gamma}, \sigma, V, \textsf{\textit{ERR}}) \in F(f) \tag{12}$$

In addition we call a global state $\sigma \in \mathit{\Sigma}$ fallible if all its contracts $\{S.C \mid \exists a \in \mathbb{B}^{160} : S = \sigma(a)\}$ are fallible.

Determinism of functionality effects In our definitions about transactions above we have defined the effects of transactions based on the state change of the initiator and actor and the fact whether they end in the end state ϵ, in another transaction T or in the error state $\textsf{ERR}$. We are not interested in the state changes of a transaction which throws an error as it will anyways end in the error state.
However for successful transactions it is important to define that if the same transaction is executed on the same global state twice, it will have the same effect both times. This property allows us to verify the outcome of the execution of a function in a smart contract. Especially for financial structures like an investment fund it can prove crucial to formally verify the outcome of certain transactions.

Definition 5 (Determinism of functionality effects). *Let $C = (a, S_V, S_F, F)$ be a contract, we call it deterministic if the following statements holds true:*

$$\begin{aligned} \forall \mathit{\Gamma}, \sigma, V, V', \forall T', T'' \in (\mathcal{T} \cup \{\epsilon\}) : \\ ((\mathit{\Gamma}, \sigma, V, T') \in F(f) \ \wedge \ (\mathit{\Gamma}, \sigma, V', T'') \in F(f)) \Rightarrow (V = V' \ \wedge \ T' = T'') \end{aligned} \tag{13}$$

In addition we call a global state $\sigma \in \mathit{\Sigma}$ deterministic if all its contracts $\{S.C \mid \exists a \in \mathbb{B}^{160} : S = \sigma(a)\}$ are deterministic.

Property preservation of transactions We have defined several properties of contracts, contract states and global states. All these properties combined ensure that the system we have created so far closely matches the behavior of the real Ethereum blockchain. The steps between global states are defined in a way so that these properties are preserved on success. In all succeeding sections if we talk about global states, we assume that these states are well-formed, fallible and deterministic.

Lemma 1 (Well-formedness preservation of transactions). *Let $\sigma \in \mathit{\Sigma}$ be a well-formed global state, let $\mathit{\Gamma} \in \mathcal{T}_{env}$ be a transaction environment and let $\tau \in \mathit{\Sigma} \cup \{\bot\}$. If $\mathit{\Gamma} \vDash \sigma \rightarrow \tau$, then $\tau = \bot$ or τ is a well-formed global state.*

Proof. We prove by induction over the derivation of $\mathit{\Gamma} \vDash \sigma \rightarrow \tau$. There are two different cases to consider, either the transaction ends in another valid well-formed global state or the transaction end in the error state.

- Error cases (FUNCTION ERROR, INSUFFICIENT BALANCE and ERROR PROPAGATION):
  If we look at the requirements for the well-formedness in Definition 3 under the assumption that the initial state was already well-formed

  $$\mathit{\Gamma} \vDash \sigma \rightarrow \bot \ \vee \ (\mathit{\Gamma} \vDash \sigma \rightarrow \sigma' \Rightarrow \sigma(a).C = \sigma'(a).C)$$

  we observe that if the transaction results in the error state $\bot$ the condition holds trivially.

- BASE STEP:
  We take a look at the relevant changes to the global state in the base step:

  $$\sigma' = \sigma\big\langle init \rightarrow \sigma(init)[b \mathrel{-}= val]\big\rangle\big\langle actor \rightarrow (b + val, V')_C\big\rangle$$

  From the first glance we recognize that only the contract states at the addresses $init$ and $actor$ change and that no contracts are modified. The only property which is not exclusively bound to the contract is the well-formedness of the contract state, see Definition 1. However we know that the original contract state is well-formed by Lemma 1 and that by Definition 2 of the well-formedness of contracts, the well-formedness of contract states is upheld for any transaction execution.

- RECURSIVE STEP:
  We assume that well-formedness is preserved in the step $\Gamma' \vDash \sigma'' \rightarrow \sigma'$, which is our induction hypothesis.
  Induction step: Here we take a look at the relevant part of the recursive step which is where the value for the new global state is set:

  $$\sigma'' = \sigma\big\langle init \rightarrow \sigma(init)[b \mathrel{-}= val]\big\rangle\big\langle actor \rightarrow (b + val, V')_C\big\rangle$$

  This update statement is identical to the one in the base step and therefore also preserves well-formedness from σ'' to σ. Consequently, by the induction hypothesis we know that the transaction $\Gamma' \vDash \sigma'' \rightarrow \sigma'$ also fulfills the well-formedness. Therefore the recursive transaction $\Gamma' \vDash \sigma \rightarrow \sigma'$ fulfills the wellformedness because σ' upholds the well-formedness.

Lemma 2 (Property preservation of transactions). *Let $\sigma \in \Sigma$ be a well-formed, fallible, deterministic global state, let $\Gamma \in \mathcal{T}_{env}$ be a transaction environment and let $\tau \in \Sigma \cup \{\bot\}$. If $\Gamma \vDash \sigma \rightarrow \tau$, then $\tau = \bot$ or τ is a well-formed, fallible, deterministic global state.*

We call the set of all well-formed, fallible, deterministic global states Σ_*.

Proof. Because the fallibility in Definition 4 and determinism in Definition 5 of global states are properties which are defined based on the underlying contracts of the global state the proof is trivial as contracts stay unchanged during execution per Definition 3. In the proof for Lemma 1 for well-formedness preservation we already covered that the contract cannot change during a transaction, because of this the fallibility and determinism is upheld.

Besides the well-formedness of contract states, contract variables can fulfill another property: Immutability. Immutable contract variables cannot be changed by any transaction, which means that they are constant since the deployment of the contract. A contract variable is immutable if no function in the smart contract can set the value of the variable. This property is commonly used in smart contracts in the real-world: A common use case is an immutable owner variable. Later on we make use of immutable contract variables to define a simple multi-signature wallet with fixed owners.

Definition 6 (Immutability of contract variables). *Let $C = (a, S_V, S_F, F)$ be a contract, we call a contract variable $(s_n, s_t) \in S_V$ at address a immutable if the following statement holds true:*

$$\begin{aligned} &\forall \Gamma, v, v' \forall \sigma, \sigma' \in \Sigma_* : \\ (s_n, v) \in \sigma(a).V \;\wedge\; (s_n, v') \in \sigma'(a).V \;\wedge\; &\Gamma \vDash \sigma \rightarrow \sigma' \Rightarrow v = v' \end{aligned} \tag{14}$$

Signatures Every address in the global state is able to sign a message (byte sequence) which produces a unique signature. We do not explicitly define how signatures are produced but simply assume that signatures share common features from modern hash algorithms like SHA-3 [46]. A signature is of type $\mathbb{B}^{256}$, therefore we also can pass

signatures as function parameters by definition of the contract variables possible values $\mathcal{V}$. Even though hashes and signatures technically allow for a collision, the chance for this negligible. We assume the hash of two randomly sampled messages x, y fulfills $P(SHA(x) = SHA(y)) \leq negl$ where SHA is a hash function and $negl$ is a negligible function.
It is additionally possible to recover the signee address a from a signature if we know the original message with the recover function denoted as $recover : \mathbb{A}_{\mathbb{B}^8} \times \mathbb{B}^{256} \rightarrow \mathbb{B}^{160}$ with

$$\forall sig\ \exists! msg : recover(msg, sig) = a \wedge \forall msg\ \exists! sig : recover(msg, sig) = a$$

The purpose of the signatures is to sign important transaction details for verification. Later on we will use the recover function to verify that a message was indeed signed by the appropriate address during execution of a transaction.

Further functionality definitions Now that we have defined the basic interaction of transactions, we can argue about the behavior of contracts by only focusing on the functionality F of a contract. The functionality defines the effect it will have on the global state, while all other parts of a transaction will stay the same for all functionalities.

5 Modelling smart contracts

Now that we have defined the smart contract semantics and provided an infrastructure to model smart contracts on, this section introduces basic, commonly used smart contract designs. The purpose is to firstly showcase how the semantics can be used to create smart contracts for structures which are already widely used, such as wallets and tokens. Secondly, these entities are used in a more advanced form in our investment fund implementation and can be used in future works as first step towards creating a fully verified financial structure.
We demonstrate this towards the end of the section by modelling one smart contract of the investment fund, a multi-signature wallet with off-chain signing to represent the wallet system used in our investment fund. This smart contract is then used to prove, in Definition 8, that the circle of trust feature, covered in Section 3.5, is indeed functional and it is impossible to send funds to untrusted wallets without permission. While we already partly introduced an owned wallet as an example in the last section, we still formally define an unowned wallet here.

Wallet Let $C = (a, S_V, S_F, F_W)$ be a contract, we call it a wallet if it has the supported functions $S_F = \{\textsc{transfer}, \textsc{receive}\}$ and if F_W is the smallest relation closed under the following rules, without ignoring the fallibility property introduced in Definition 4:

$$\textsc{receive}\ \frac{\mathit{\Gamma}.T = (\mathit{init}, \mathit{actor}, \textsc{receive}, [\,], \mathit{val}) \qquad \sigma(\mathit{actor}) = (b, V)_C}{(\mathit{\Gamma}, \sigma, V, \epsilon) \in F_W(\textsc{receive})}$$

With the rule above we can observe that the RECEIVE function is a simple function which only ensures that Ether can be transferred to it. It fails only if the initiator does not have sufficient balance to transfer the value. In Solidity this function can represent an empty, payable, default function. A function in Solidity is called payable if it allows to receive Ether as part of the execution.

The TRANSFER function makes use of the RECEIVE function to initiate a transfer of Ether.

$$\textsc{transfer}\ \frac{\begin{array}{c} \mathit{to} \in \mathbb{B}^{160} \qquad \mathit{amount} \in \mathbb{N} \qquad \mathit{\Gamma}.T = (\mathit{init}, \mathit{actor}, \textsc{transfer}, [\mathit{to}, \mathit{amount}], 0) \\ \sigma(\mathit{actor}) = (b, V)_C \qquad T' = (\mathit{actor}, \mathit{to}, \textsc{receive}, [\,], \mathit{amount}) \end{array}}{(\mathit{\Gamma}, \sigma, V, T') \in F_W(\textsc{transfer})}$$

This theoretical smart contract of wallet models the bare minimum for a smart contract to manage the native currency Ether, at least in the high-level setting that we have defined our semantics in. This setting closely resembles how smart contracts are programmed (in e.g., Solidity), but has some differences compared to the final EVM bytecode. It is important to note that this contract has no implementation of ownership which means anyone could send currency from this wallet.

Token Let $C = (a, S_V, S_F, F_{tok})$ be a contract. We call C a *token* if it has at least two class variables $(\text{TBAL}, \mathit{bitstring}^{160} \mapsto \mathit{natural}), (\text{TSUPPLY}, \mathit{natural}) \in S_V$ (short for token balance and token supply), exactly one supported function $S_F = \{\text{TRANSFERT}\}$ (short for transfer tokens) and the F_{tok} is the smallest relation closed on the rule introduced below, without ignoring the fallibility property introduced in Definition 4. We call the set of all tokens $\mathbf{T}$.

The TRANSFERT function abides rules when executed by a transaction, we present the effect here. It should be noted that this definition is compatible with the transfer function interface of the ERC20-Token Standard, introduced in Section 2.2.

$$\text{TRANSFERT}\quad \frac{\begin{array}{c} \mathit{to} \in \mathbb{B}^{160} \qquad \mathit{amount} \in \mathbb{N} \qquad \mathit{\Gamma}.T = (\mathit{init}, \mathit{actor}, \text{TRANSFERT}, [\mathit{to}, \mathit{amount}], 0) \\ \sigma(\mathit{actor}) = (b, V)_C \qquad (\text{TBAL}, v_{\text{TBAL}}) \in V \qquad v_{\text{TBAL}}(\mathit{init}) \geq \mathit{amount} \\ v'_{\text{TBAL}} = v_{\text{TBAL}}\langle \mathit{init} \rightarrow v_{\text{TBAL}}(\mathit{init}) - \mathit{amount}\rangle\langle \mathit{to} \rightarrow v_{\text{TBAL}}(\mathit{to}) + \mathit{amount}\rangle \\ V' = V[(\text{TBAL}, v_{\text{TBAL}}) \rightarrow (\text{TBAL}, v'_{\text{TBAL}})] \end{array}}{(\mathit{\Gamma}, \sigma, V', \epsilon) \in F_{tok}(\text{TRANSFERT})}$$

Here we specify the most basic components of a token, introduced in Section 2.2, built on top of an existing network. A token has a registry of all balances TBAL of which the values can be altered by transferring tokens to other addresses. The TSUPPLY variable is a counter for the total balances across all address balances of TBAL. The total supply is often an immutable variable and is fixed from the token launch if the token has a fixed supply. However some tokens, such as the token used in our investment fund changes the total supply by minting and burning new tokens on demand.
Holders of tokens can call the TRANSFERT function to transfer some of their token holdings to a different address. This transfer is similar to the basic value transfer in the wallet construct with the only difference being that the token balance instead of the native balance is used.
A token as specified here is not a fully-fledged ERC20-Token by definition, but is compatible with ERC20. Besides a transfer function and a total supply variable, a fully-fledged ERC20-Token also has extra functionality to provide token holders with the possibility to authorize other addresses to send tokens on their behalf. We do not define this explicitly here because these functions are only indirectly relevant for the investment fund.

Token-compatible Wallet Let $C = (a, S_V, S_F, F_{TW})$ be a contract. We call it a *token-compatible wallet* if it has the supported functions
$S_F = \{\text{TRANSFER}, \text{RECEIVE}, \text{REQTTRANS}\}$ and implements $F_{TW} \supseteq F_W$. That means compared to the normal wallet it has one additional function called REQTTRANS (short for *request token transfer*). This function is used to transfer tokens to a different wallet. Technically the tokens are not held by the wallet, but rather the wallets balance of a certain token is stored in the TBAL variable of the token contract state. For this reason the function is prefixed with *request*, in order to signalize that the token contract will execute the transfer and not the wallet itself.

REQTTRANS

$$\frac{\begin{array}{c} tok \in \mathbf{T} \\ to \in \mathbb{B}^{160} \qquad amount \in \mathbb{N} \qquad \mathit{\Gamma}.T = (init, actor, \text{REQTTRANS}, [tok.a, to, amount], 0) \\ \sigma(actor) = (b, V)_C \qquad T' = (actor, tok.a, \text{TRANSFERT}, [to, amount], 0) \end{array}}{(\mathit{\Gamma}, \sigma, V, T') \in F_{TW}(\text{REQTTRANS})}$$

Contract Ownership The difference between an owned contract and a normal contract is that not every participant on the blockchain can execute all functions of the contract, but rather only an owner or a set of owners. An owned contract is a commonly used concept on the blockchain and can represent the idea of what most people imagine by the concept of a wallet, in a sense that only owners can use the balance of a wallet. Here we only offer a template on how ownership could be uniformly specified across contracts while not going into detail about security properties. Later in Section 5 we will formally define and prove ownership of a specific wallet contract.

In order to get to the concept of an *owned* contract, we first have to define necessary characteristics of a contract to offer ownership features. Let $C = (a, S_V, S_F, F)$ be a contract, which has at least four class variables

$$\begin{aligned} &(\text{OWNERS}, Array(bitstring^{160})), (\text{THRESHOLD}, natural), \\ &\qquad\qquad\qquad\qquad (\text{NONCE}, natural) \in S_V. \end{aligned} \tag{15}$$

Furthermore let OWNERS and THRESHOLD be immutable. The OWNERS array saves the addresses which share ownership of the contract. The THRESHOLD value states how many owners need to allow a function call by signing it. If the THRESHOLD value is bigger than 1, we also call an $owned$ contract a *multi-signature* contract instead, which is a common term in the blockchain sector and simply states that multiple signatures are needed to operate the contract. Lastly the NONCE is used as a standard cryptographic nonce in order to guarantee unique signatures for every call the contract will make.

All of the above mentioned class variables are used to restrict access to the contract. This protection must happen in the definition of how the functionality F of a contract can arrive at a successful result (without a runtime error ERR). For this purpose we now present a set of rules which can be inserted into a contract's functionality to protect functions with ownership.

OWNERSHIPTEMPLATE

$$\frac{\begin{array}{c} sigs \in A_{\mathbb{A}_{\mathfrak{S}}} \\ data \in \mathcal{V} \qquad \mathit{\Gamma}.T = (init, actor, f, [sigs, data], val) \qquad \sigma(actor) = (b, V)_C \\ (\text{THRESHOLD}, v_{\text{THRESHOLD}}), (\text{OWNERS}, v_{\text{OWNERS}}), (\text{NONCE}, v_{\text{NONCE}}) \in V \\ msg = actor^\frown f^\frown data^\frown v_{\text{NONCE}} \\ \forall sig \in sigs, \exists owner \in v_{\text{OWNERS}} : owner = recover(msg, sig) \\ |sigs| > 1 \Rightarrow (\forall i, j \in \mathbb{N}, i < j < |sigs| : sigs[i] \neq sigs[j]) \qquad |sigs| \geq v_{\text{THRESHOLD}} \end{array}}{ownershipTemplate(V, \mathit{\Gamma})}$$

The construct above describes the concept of a multi-signature contract. We can observe that msg uniquely identifies the transaction via the wallet address $actor$, function execution details $f, data$ and nonce v_{NONCE}. Owners of the multi-signature wallet are required to sign the message off-chain which can reduce the possibility of a smart contract vulnerability (see Subsection 2.3) and is usually not done manually but with client programs. The set of signatures can then be sent from any address to the wallet, however the transaction will only be successful if the signatures match the purpose (address, function, parameters and nonce) of the transaction. We check that every signature must be created by an owner and we verify that each signature is unique and therefore from a unique owner. This way it also does not matter if the OWNERS array of the wallet has duplicate entries, because we verify that the signatures are unique. Lastly we also ensure that enough signatures have been provided to match the threshold of the wallet.

One thing which is not possible to encode in the rule above is the needed increment of the nonce. The nonce is a vital part of the unique identification of a transaction in case a transaction with exactly the same parameters is executed twice. The nonce has to be incremented after every successful execution of a function and has to be saved in the global state. This incrementation of the nonce has to be done explicitly by a contract's functionality if it implements this type of ownership and will be modelled faithfully for concrete contracts.
This implementation of ownership uses off-chain signatures because the signatures are passed as function parameters and are created off-chain. A more common approach to ownership is on-chain signing. In order to confirm a transaction which needs e.g., three signatures using on-chain signatures, three owners would send three separate transactions from their addresses which act as signatures. The upside of this is that it is easier to use (no need for a software to sign off-chain) and it decreases code complexity, however the transaction cost is much higher especially if the owner count is high, because a separate transaction is needed for every signature.
For the definition of a multi-signature wallet above we define the parameters of our transaction as an array of signatures and data, which are the rest of the parameters stored in a byte-array. The message is defined as all relevant data of the transaction, consisting of the actor, the function, the data and the nonce. This structure of messages and signing transactions follows an Ethereum standard under EIP191 [65].

Internal and external owners Internal and external owners are used to reflect a feature of the investment fund which we mentioned as *circles of trust* in Section 3.5. This classification of owners can be used to protect certain functions of a smart contract with additional security by requiring external owners (e.g., stake-holders) to approve the execution. A use case for this would be to allow internal owners to transfer currency between trusted wallets, but prevent them from moving currency to unknown, potentially malicious, wallets. Naturally this feature reminds of a simplified version of an owner hierarchy and could be extended to not only include two levels of owners, but an arbitrary amount.

The addresses of OWNERS of an owned wallet can be split into $internal$ addresses A_I and $external$ universally trusted addresses A_E, so that $A_I \cap A_E = \emptyset$. We will use these definitions below by introducing different levels of owners and different levels of permissions. Furthermore we want to differentiate between transactions which are only signed by internal addresses and those which also include external addresses as signees.

Definition 7 (Internal signature). *Let $C = (a, S_V, S_F, F)$ be an owned contract, let A_I be its internal addresses and let $sig \in \mathfrak{S}$ be a signature. We call sig $internal$ to C if it fulfills:*

$$\exists a_I \in A_I \; \exists msg : recover(msg, sig) = a_I$$

Consequently we call an array of signatures $sigs$ with $sigs \in \mathbb{A}_{\mathfrak{S}}$ internal to a contract if all its signatures are internal to the contract.

In the following we introduce a specific multi-signature wallet contract, which is modelled after the wallet system used by the investment fund covered in Section 6. Furthermore we introduce security properties on it and prove them in Definition 8.

Restricted Wallet Let $C = (a, S_V, S_F, F_R)$ be a contract which has at least four class variables

$$\begin{gathered}(\textsc{owners}, Array(bitstring^{160})), (\textsc{threshold}, natural),\\ (\textsc{reqsig}, bitstring^{160} \mapsto natural), (\textsc{nonce}, natural) \in S_V.\end{gathered} \tag{16}$$

Additionally let OWNERS, THRESHOLD and REQSIG be immutable. The OWNERS, THRESHOLD and NONCE variables fulfill the same purpose as in the owned contract. The variable named REQSIG stands for the required signatures needed to execute a transfer to a certain address. This new variable is used to introduce a different threshold for different addresses.
Let the set of supported functions be $S_F = \{\text{R}\textsc{transfer}, \text{RR}\textsc{eq}\text{T}\textsc{rans}, \textsc{receive}\}$, without ignoring the fallibility property introduced in Definition 4, and let F_R be the smallest relation closed under these rules. The RTRANSFER function (short for restricted transfer) is an adaption from the TRANSFER function combined with a similar property as $ownershipTemplate$ with a variable threshold. RREQTTRANS (short for restricted request for token transfer) is also a stricter version of REQTTRANS.

RTRANSFER

$$\frac{\begin{gathered}sigs \in \mathbb{A}_{\mathfrak{S}} \qquad to \in \mathbb{B}^{160} \qquad amount \in \mathbb{N}\\ \mathit{\Gamma}.T = (init, actor, \text{R}\textsc{transfer}, [sigs, to, amount], 0) \qquad \sigma(actor) = (b, V)_C\\ (\textsc{threshold}, v_{\textsc{threshold}}), (\textsc{owners}, v_{\textsc{owners}}), (\textsc{reqsig}, v_{\textsc{reqsig}}), (\textsc{nonce}, v_{\textsc{nonce}}) \in V\\ msg = actor^\frown f^\frown to^\frown amount^\frown v_{\textsc{nonce}}\\ \forall sig \in sigs, \exists owner \in v_{\textsc{owners}} : owner = recover(msg, sig)\\ |sigs| > 1 \Rightarrow (\forall i, j \in \mathbb{N}, i < j < |sigs| : sigs[i] \neq sigs[j])\\ |sigs| \geq ((v_{\textsc{reqsig}}(to) = 0) \; ? \; \textsc{threshold} : v_{\textsc{reqsig}}(to))\\ T' = (actor, to, \textsc{receive}, [\,], amount)\\ V' = V[(\textsc{nonce}, v_{\textsc{nonce}}) \rightarrow (\textsc{nonce}, v_{\textsc{nonce}} + 1)]\end{gathered}}{(\mathit{\Gamma}, \sigma, V', T') \in F_R(\text{R}\textsc{transfer})}$$

$$\textsc{RreqTtrans}\quad \frac{\begin{array}{c} sigs \in \mathbb{A}_{\mathfrak{S}} \qquad tok \in \mathbf{T} \qquad to \in \mathbb{B}^{160} \\ amount \in \mathbb{N} \qquad \mathit{\Gamma}.T = (init, actor, \textsc{RreqTtrans}, [sigs, tok.a, to, amount], 0) \\ \sigma(actor) = (b, V)_C \qquad msg = actor^\frown f^\frown a_{tok}{}^\frown to^\frown amount^\frown v_{\textsc{nonce}} \\ (\textsc{threshold}, v_{\textsc{threshold}}), (\textsc{owners}, v_{\textsc{owners}}), (\textsc{reqsig}, v_{\textsc{reqsig}}), (\textsc{nonce}, v_{\textsc{nonce}}) \in V \\ \forall sig \in sigs, \exists owner \in v_{\textsc{owners}} : owner = recover(msg, sig) \\ |sigs| > 1 \Rightarrow (\forall i, j \in \mathbb{N}, i < j < |sigs| : sigs[i] \neq sigs[j]) \\ |sigs| \geq ((v_{\textsc{reqsig}}(to) = 0) \text{ ? } \textsc{threshold} : v_{\textsc{reqsig}}(to)) \\ T' = (actor, tok.a, \textsc{transferT}, [to, amount], 0) \\ V' = V[(\textsc{nonce}, v_{\textsc{nonce}}) \rightarrow (\textsc{nonce}, v_{\textsc{nonce}} + 1)] \end{array}}{(\mathit{\Gamma}, \sigma, V', T') \in F_R(\textsc{RreqTtrans})}$$

We call a contract with the above stated properties a *restricted wallet*. Moreover, we call an address a_{Tr} a $trusted$ address if $\textsc{reqsig}(a_{Tr}) \leq |A_I|$ which means it can be signed by exclusively internal owners. Otherwise we call it $untrusted$. The set of $trusted$ addresses of a restricted, owned wallet is referred to by A_{Tr}.

If we observe the definition above, we can see that it is not all too different from a standard multi-signature wallet. It still shares all its contract variables, namely OWNERS, THRESHOLD, and NONCE. However by defining a dynamic threshold as REQSIG restricted wallets offer the implementation of trust circles. With this it is possible to define the set of trusted wallets for an investment fund. It is intended to include all addresses to the set of trusted wallets which are needed to operate on a normal basis. This can include internal wallets, partner wallets and exchange wallets. Sending Ether or tokens to an unknown wallet requires extra permissions from a different set of owners. This is not only useful to provide increased trust for shareholders, but also prevents accidental or malicious calling of an untrusted address.

This implementation of a wallet fulfills the property of *circles of trust* which was described in Section 3.5. In the following we provide a fitting property and prove that it holds on restricted wallets. The intuition behind this property is that if a restricted wallet is called by only internal owners, then it is impossible to transfer Ether or tokens to unknown/untrusted wallets. This is expressed by stating that should the wallet receive any transaction with a set of exclusively internal signatures, then the Ether or tokens transferred will be transferred to a trusted wallet and therefore stay in its circle of trust.

Definition 8 (Balance preservation in circles of trust). *Let $C = (a, S_V, S_F, F_R)$ be a restricted wallet with its trusted addresses A_{Tr}. Let $\mathit{\Gamma}$ be a transaction environment where $\mathit{\Gamma}.T$ is a signed transaction with an array of internal signatures $sigs$. We say C preserves its balances within its circles of trust if it fulfills the following criteria.*

$$
\begin{aligned}
&\forall \sigma, \sigma' \in \Sigma_* \; \forall \Gamma \; \forall tok \in \mathbf{T}, \Gamma.T = (init, a, f, [sigs, data], T') : \\
&\Gamma \vDash \sigma \to \sigma' \;\wedge\; (\textsc{TBal}, v) \in \sigma(tok.a) \;\wedge\; (\textsc{TBal}, v') \in \sigma'(tok.a) \Rightarrow \\
&\qquad v(a) + \sum_{a_{Tr} \in A_{Tr}} v(a_{Tr}) = v'(a) + \sum_{a_{Tr} \in A_{Tr}} v'(a_{Tr}) \\
&\qquad \wedge\, \sigma(a).b + \sum_{a_{Tr} \in A_{Tr}} \sigma(a_{Tr}).b = \sigma'(a).b + \sum_{a_{Tr} \in A_{Tr}} \sigma'(a_{Tr}).b
\end{aligned}
\tag{17}
$$

Intuitively described the definition above formulates that a contract which upholds the balance preservation within its circles of trust cannot transfer any money outside the trust circle without external signatures. The criteria specify that after any successful transaction execution the sum of balances of trusted addresses (both Ether and token balances) is guaranteed to be equal to the sum of balances before the transaction. This property is later on used in the investment fund to provide safety for stakeholders.

Theorem 1. *Every restricted wallet preserves its balances within circles of trust.*

Proof. By Definition 8 we have the premise that the definition only applies for every successful transaction between two well-formed, fallible and deterministic global states $\Gamma \vDash \sigma \to \sigma'$. Another restriction is that the actor of the transaction $\Gamma.T$ has to be the restricted wallet itself. By the definition of the restricted wallet we know that there are only two possible functions which can have a successful execution: RTRANSFER and RREQTTRANS, see Equation 5.

- Case RTRANSFER: By the immediate definition of RTRANSFER we observe that no value is transferred to the restricted wallet by looking at its transaction and the responding step:

  $$\Gamma.T = (init, actor, \textsc{RTransfer}, [sigs, to, amount], 0)$$

  This transaction is only executable by the RECURSIVE STEP (see Section 4.3), because it executes a sub-transaction RECEIVE afterwards and does not result in the error state. By Section 4.3 the above mentioned value of 0 is transferred and the contract state is changed according to the functionality. That means no value is transferred and the current transaction already fulfills a part of the balance preservation because $\forall a : \sigma(a).b = \sigma'(a).b$. The only contract variables relevant to the balance preservation property of the restricted wallets are the states of tokens, as we do not interact with any tokens this first recursive call does not violate the property. The RECURSIVE STEP arrives at its intermediary state σ'' and executes the sub-transaction of RTRANSFER with $\Gamma' \vDash \sigma'' \to \sigma'$

  The sub-transaction is specified in the definition of RTRANSFER as

  $$T' = (actor, to, \textsc{receive}, [\,], amount)$$

  If we look at the RECEIVE function in Section 5 we can see that it is only successfully executable by the BASE STEP because it has no sub-transaction. Furthermore the function does not change the contract variables of any contracts as per definition it returns the same set of contract variables V the receiver to had assigned before the transaction. By the definition of the BASE STEP the remaining changes are that $amount$ is transferred from the $actor$ address (the restricted wallet) to the address to. This is the only possible value transfer and could only potentially violate the balance preservation clause

$$\sigma(a).b + \sum_{a_{Tr} \in A_{Tr}} \sigma(a_{Tr}).b = \sigma'(a).b + \sum_{a_{Tr} \in A_{Tr}} \sigma'(a_{Tr}).b$$

if to is not a trusted address (by Definition 8).

Now we can make a trivial proof by contradiction to show that to has to be a trusted address. We know by Equation 5 of RTRANSFER that an untrusted wallet has a higher signature requirement than internal owners exist $\text{REQSIG}(a_{untrusted}) > |A_I|$. By the definition of balance preservation the property only applies to signed transactions with an internal array of signatures which means it is impossible to fulfill the clause

$$|sigs| \geq ((v_{\text{REQSIG}}(to) = 0) \text{ ? THRESHOLD} : v_{\text{REQSIG}}(to))$$

because any untrusted wallet needs a greater number of signatures than there are internal addresses.

The other clause of balance preservation

$$v(a) + \sum_{a_{Tr} \in A_{Tr}} v(a_{Tr}) = v'(a) + \sum_{a_{Tr} \in A_{Tr}} v'(a_{Tr})$$

only refers to the balances of tokens, as we do not interact with tokens it continues to hold.

This means the balance preservation holds on RTRANSFER because it is only possible to transfer value and no tokens, and this value can only be transferred to trusted wallets.

- Case RREQTTRANS:

  This case is mostly similar to the previous one. We can see that RREQTTRANS, defined in Equation 5, and its sub-transaction both have a value of 0. That means the second part of the balance preservation property holds as all balances stay unchanged. The subtransaction of RREQTTRANS is

  $$T' = (actor, tok.a, \text{TRANSFERT}, [to, amount], 0)$$

  If we look at the definition of TRANSFERT in Section 5 we see that the token at address $tok.a$ moves $amount$ tokens from the initiator of the transaction to the address to. This has the following effect on the token's global state:

  $$V' = V[(\text{TBAL}, v_{\text{TBAL}}) \rightarrow (\text{TBAL}, v'_{\text{TBAL}})]$$

  Where v'_{TBAL} contains the new balances of the sender and recipient and overwrites the old balances. That means we fulfill the balance preservation if to is a trusted wallet. We can argue analogous to as we did for RTRANSFER that to can only be trusted by Definition 8.

Further work The structures we have introduced and defined in this section should serve as a demonstration on how to formalize common blockchain smart contracts and how to describe interactions between them. The provided specifications can easily be extended to provide more functions to structures like the wallet or token. While we have only used manual proofs to verify some properties, it should be possible to transform this semantics so that it would fit in a proof assistant which is potential further work.
Furthermore with the restricted wallet and its proof regarding *balance preservation* we provide a link to the practical implementation of the investment fund which uses the same feature for its wallets. Even though the semantics cannot offer a clear link between itself and Solidity (because Solidity itself does not offer semantics) it can be seen as the first step towards working on high level semantics which can be applied to programming languages.

6 Investment fund implementation

In the course of this thesis we have developed a fully-fledged investment fund software *ERCFund* [60] built purely on the Ethereum blockchain for managing ERC20 tokens. ERCFund makes it possible to invest into an actively managed portfolio of ERC20 tokens and Ether by introducing an on-demand minted and burned token as the medium for shares in the fund. Compared to some other closed-end funds (e.g., TaaS.fund, The Token Fund) you can buy shares/tokens at any time by simply sending Ether to the fund. These shares/tokens can also be sold at any time by calling the withdraw function of the fund which sends the corresponding value of Ether back to a specified wallet.

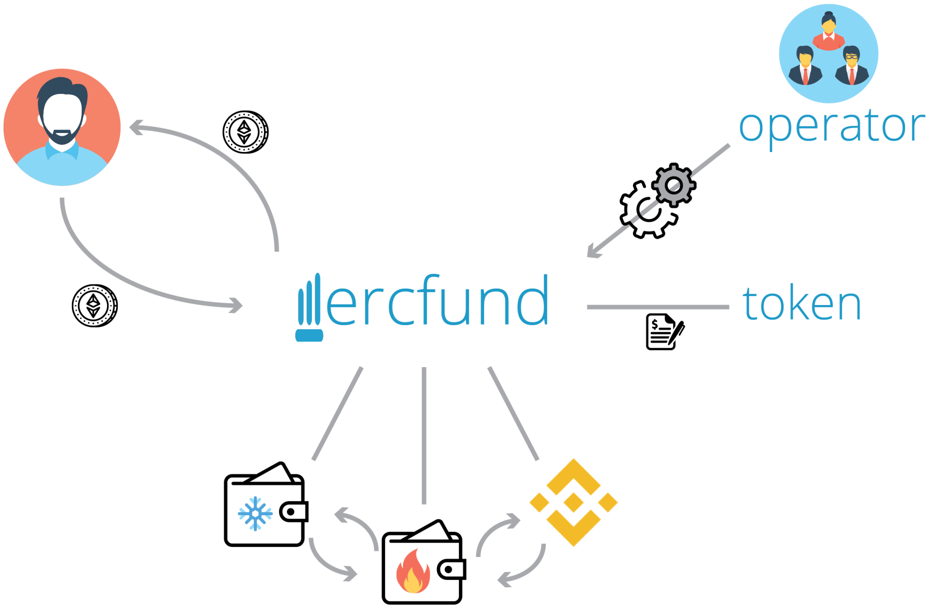

Fig. 8: Illustration of the architecture and functionality of the ERCFund software.

Fund managers can freely trade with all currencies and make profit. Based on the assets under management the price for one token should be continuously updated. ERCFund comes with a multi-signature implementation for a fund operator contract. This implementation is especially designed for the purpose of managing a fund. It offers cold-wallet support and a defined set of trusted wallets for different trust levels. Additionally the multi-signature operator has a significantly lower gas cost than traditional multi-signature wallets by moving signing transactions off-chain. In Figure 8 the basic architecture and functions of the investment fund are illustrated to give quick, intuitive overview.

For smart contracts it is essential to make sure that the code is bug-free if large amounts of money are handled by them. To achieve a high code quality and security we reused as much community-reviewed code as possible, such as token contracts from OpenZeppelin [23], which are already tested and in use by a large number of people. All the code written by us is rigorously tested: While the total lines of code are less than a thousand, we provide 127 tests covering all functionalities and achieve a code coverage of 100% and a branch coverage of 96%.
In order to use the suite of smart contracts for investment funds we imagined that they would be used with a client side program fitting the needs of the fund managers. While it is absolutely possible to use all features of the fund without it, it would be difficult and time consuming to sign transactions manually. Additionally some features, like price updates, need to happen in small regular intervals. Which is why it is infeasible to provide price updates manually.
In this section we describe the architecture and implementation in detail and provide code examples for interesting and essential parts of the investment fund.

6.1 Architecture

The software is split into four different layers where each of them fulfills a distinct function and part of the investment fund. Every layer is represented by separate smart contracts deployed on the blockchain which interact with each other. Many of the features implemented in the software are conceptually described in Subsection 4.1. Moreover the section also reasons which features are needed to match functions of a traditional investment fund and how these features can be implemented in smart contract form.

Wallet - Layer The wallet layer represents the lowest layer of the software and is used to store the currency of the investment fund. The layer does not necessarily consist of only one wallet but can be a set of arbitrary many wallets. What might seem counter-intuitive at first is that the fund contract itself is also a fund wallet. The fund contract needs to hold Ether in order to liquidate shares on demand, for this reason the functions in the wallet contract are adapted to work with the fund contract as well.
The wallet contract is not vastly different from what is commonly used for wallets. It implements ownership so that it can only be accessed by the fund contract which is achieved with a function modifier. Function modifiers are pieces of code which can be inserted in the beginning of a function by including their signature in the function signature. An example is given in Figure 9, here the owner of the contract is specified in the constructor and checked whenever the modifier is called.

In order to send and receive Ether and send tokens, the wallet contract contains a payable fallback function. A fallback function is called when no or a non-existent function signature is called. Payable in this context means that the function is accepting a value transfer (in the form of Ether) during its execution. Besides emitting an event, the fallback function is empty and is only used to receive Ether. The function `sendEther` is used to move Ether from the wallet to another address and is only executable by the fund contract, its owner. Lastly the `sendTokens` function takes a token address as an

```
modifier onlyOwnerOrInternal() {
    require(msg.sender == owner || msg.sender == address(this));
    _;
}
```

Fig. 9: Ownership modifier in the Fund Wallet contract.

input and moves the specified amount of tokens to another address. The implementation of these functions is basic but will be partly covered in Subsection 6.2.

Token - Layer As we already discussed before, tokens are used to represent shares of the investment fund. The token is directly bound to the fund contract, this contract is both issuing new tokens and liquidating existing tokens. We will not go into depth about the standard ERC20 functions the token implements here or in the implementation section. But besides these functions the token is extended with a minting and burning feature. This means that the total supply of tokens is not fixed but can fluctuate during the contract's lifetime. Similarly to the wallet contract, the token contract is also owned by the fund contract and only allows the fund contract to create new tokens and burn existing ones.

Fund - Layer The fund layer acts as the connection between all other layers and implements most of the crucial features. The smart contract itself extends from the wallet - layer and a common implementation of the pausable pattern.
Pausable means that functions of the contracts can be restricted if the fund is paused. This is a common design pattern in smart contracts as it can help maintain the smart contract. For example, in an investment fund one strategy to change the buying and selling price of a share could be to pause all purchases and sales, change the price and reactivate the features again. This makes it impossible for shareholders to frontrun any prices change. If e.g., the price of a share is changed from 1 Ether to 0.9 Ether, shareholders could detect this price change before it actually happens because for a few seconds the transaction will be in a pending status until a miner processes it. By running a node themselves, they scan pending transactions and in case a price change is detected a transaction is started with a very high gas price to sell shares. Because miners will generally try to maximize their profits, they will process transactions with a high gas price first as they are more lucrative. Consequently shareholders could buy or sell right before a price change and profit while taking no risk.

The fund features include most functions potential customers can interact with. Namely these are the `withdrawTo` and `buyTo` functions. The withdrawTo function enables shareholders to liquidate their shares/tokens. The amount of Ether received in exchange for tokens is calculated based on the current price and withdraw fee. In the case of a successful withdrawal the tokens received are burned.
The buyTo function takes the role of creating new shares/tokens if the demand is higher

than the amount of already existing tokens. Similarly the caller can send Ether and new tokens are minted based on the current price and fees and attributed to the receiver.

Price updates are conducted via a price update method which can only be called by the owner/operator contract. Pricing is currently implemented in a simple way: Only one current price contract variable exists which is both used for buying and selling shares. Because of the nature of the blockchain and constant price fluctuations it is impossible to save a current, completely accurate value of a share. With there being only a single price variable, it is clear that small price inaccuracies can be abused by either selling or buying shares. In its current form arbitrage (selling and buying of the same asset in different markets for a profit) is hard to deal with, but is reduced in effectiveness by implementing a withdraw- and purchase-fee. Depending on how high these fees are set arbitrage possibilities could be reduced.

A different approach to combatting arbitrage would be to implement a purchase and withdraw price separately with them being able to contain a variable buffer as a counteraction. While the approaches seem similar, the latter could potentially offer greater flexibility and can more easily be adapted based on the volatility of the market. In any case, it is necessary to update the prices in regular, short intervals depending on the volatility.

Internal management of Ether and tokens also happens directly through the fund, however the functions to move assets can only be accessed through the fund operator in which a more complicated procedure is used based on senders and recipients.

```
contract Pausable is Ownable {

  bool public paused = false;

  modifier notPaused() {
    require(!paused);
    _;
  }

  function pause() onlyOwner public {
    paused = true;
  }

  function unpause() onlyOwner public {
    paused = false;
  }
}
```

Fig. 10: Implementation of a pausable smart contract to be inherited by other smart contracts.

Operator - Layer The operator layer is a multi-signature contract on top of the fund class in order to introduce a layer of security. It is different from other multi-signature wallets (like the Gnosis MultiSigWallet [9]) by signing transactions off-chain. Every function of the contract requires a set of signatures passed with it, besides the normal parameters. Before the action is performed, it is checked if each signature contains all information about the transaction. Signing on-chain has pros and cons, the main pro is that it is simpler to use because you do not need a special application to sign your transactions off-chain. The cons are that transactions are more expensive, because you need to send a transaction from every owner. An actively managed fund might need to make hundreds of transactions every day, signing off-chain decreases transaction cost significantly.

As by the nature of multi-signature contracts the operator saves a set of owners at its creation. These owners are split into two different groups: The fund managers and the trust party. The different sets of owners enable us to create two different layers of trust, see Section 3.5. As already mentioned, members of the trust party group could be e.g., an external audit firm, a group of significant investors or generally trusted personalities. Depending on the actions called in the operator contract either only signatures of the fund managers are needed or signatures of both groups are needed. The motivation behind the feature for different circles of trust is to increase the transparency and offer an easy tool to enable lesser known investment managers to increase trust from the general public by adding another layer of security. However it is clear that this feature is not integral for the functionalities of an investment fund and might not be interesting to certain groups. For this reason the trust party/circle of trust feature is implemented in a way which makes it completely optional.

The operator contract adds wrapper functions for all functions of the fund contract which should not be accessible by the general public. Additionally it implements a set of trusted wallets. New addresses can be added to the trusted list by signing the transaction with the trust party. All internal wallets are automatically added as a trusted wallet, which makes transfers between them require less permissions.
Furthermore the operator class implements the addition of cold wallets to the fund, which are especially safe wallets which were never connected to the internet. Cold wallets can be added because the operator class runs with off-chain signatures which means the cold key can sign transactions without being connected to the blockchain or internet.

6.2 Implementation

We have summarized the key features and the overall architecture of the investment fund in Subsection 6.1. As opposed to viewing code again on the levels of different layers, the approach used here will be feature-driven. Instead of analyzing important code of the fund contract we will cover a full execution throughout multiple contracts. It is not possible to cover the complete implementation in this section but it is publicly available on Github [60].

Share/Token purchase and withdrawal Arguably the central part of our investment fund is the token system which represents shares within the fund. This system is unique to our knowledge in a way that it creates new shares and liquidates shares on demand. The fund contract implements two functions, see Figure 11, to support this feature which can be called by share-holders or potential customers.

In order to purchase shares an interested party would send the amount of Ether with which it wants to buy shares to the fund contracts `buyTo` function (line 3) or the default function (which simply redirects to the former function). By observing the function signature (line 3) we see that it takes a single argument, the `_to` address to which the shares will be attributed to. In case the default function is called, this address defaults to the message sender. Furthermore both the withdraw and purchase function signatures have a number of modifiers(e.g., line 3) appended to them, we recall that modifiers are functions which get executed before the actual function body. They are mostly one-liners to check that the input and the fund contract are in a valid state. The modifiers `external` and `public` respectively state that the function can either only be called by external addresses or by external addresses and internal functions. The three functions `hasToken`, `priceSet` and `whenNotPaused` check the fund contract's state. For a successful purchase the fund has to be correctly initialized with a token and has to have a set price per token, additionally the fund cannot be paused (covered in Section 6.1). Finally `notNull` makes sure that the address specified to receive the shares is not the zero address. This is a common design pattern because if a function is accidentally called without adding arguments, it can happen that the function parameters are filled with zeros which would result in the shares being lost.

In line 5 the value sent with the execution is converted based on the current price of a share and subsequently in line 6 the purchase fee is subtracted. The current price is split into numerator and denominator as Solidity does not support floating point numbers. Once the amount of tokens to create is calculated the fund contract calls the `mint` function of the token contract in line 7 and specifies the address to which the newly minted tokens belong to. If this is successful an event is fired in the following line to notify any contract listeners.

The withdraw function works similarly, however the message sender can only withdraw shares/tokens attributed to their address. Instead of sending the value directly with the transaction, which is not possible for tokens, the sender specifies the amount of tokens they want to withdraw. After checking if the requestor has the amount of tokens they want to exchange in line 13, the value is converted to Ether the same way as in the buyTo function. Lastly, the contract burns the amount of tokens (line 18) and transfers the appropriate amount of Ether to the receiver in line 19. It can happen that this transaction fails because the fund contract needs to have sufficient Ether available to liquidate the shares. In times of high volatility or when a large stakeholder wants to cash out, the contract might not be able to fulfill the transfer. In this case an event is emitted to signal the failure in line 22. For fund managers to combat withdrawal failure it can be necessary to keep direct contact with large stakeholders and make them announce a large withdrawal. For the case of many small withdrawals an off-chain program can observe the remaining balance and automatically counteract withdrawals to a certain

```
contract Fund is FundWallet, Pausable {
...
    function buyTo(address _to) public payable hasToken whenNotPaused priceSet notNull(_to) {
        require(msg.value != 0);
        uint256 convertedValue = msg.value.mul(currentPrice.numerator).div(currentPrice.denominator);
        uint256 purchaseValue = convertedValue.mul(PURCHASE_FEE).div(100);
        token.mint(_to, purchaseValue);
        emit Purchase(msg.sender, _to, purchaseValue, msg.value);
    }

    function withdrawTo(address _to, uint256 _value) external hasToken whenNotPaused priceSet notNull(_to) {
        require(_value != 0);
        require(token.balanceOf(msg.sender) >= _value);
        address requestor = msg.sender;
        uint256 convertedValue = currentPrice.denominator.mul(_value).div(currentPrice.numerator);
        uint256 withdrawValue = convertedValue.mul(WITHDRAW_FEE).div(100);
        if (address(this).balance >= withdrawValue) {
            token.burn(requestor, _value);
            _to.transfer(withdrawValue);
            emit Withdrawal(requestor, _to, _value, withdrawValue);
        } else {
            emit FailedWithdrawal(requestor, _to, _value, withdrawValue);
        }
    }
...
}
```

Fig. 11: Withdrawal and purchase functions of the investment fund.

extend by automatically sending Ether from other internal wallets if the balance gets too low.

Token creation and burning While the customers only interact with the fund contract, the minting and liquidating of shares/tokens happens in the token contract. We already discussed the basics of tokens in Section 2.2. A mintable token has no fixed total supply by design and should have defined methods on how new tokens can enter circulation. In the case of this investment fund, the `mint` function of the token in Figure 12 is only callable by its owner, the fund contract, which is only calling the mint function in the purchase function.
In line 3 the minting process takes as an input the amount of tokens to mint and the address to which the tokens will be minted to. Next, in lines 4 and 5, the total supply and the balance of the receiver is increased accordingly. Lastly two events are fired in lines 6 and 7 which both signalize the same minting process, because there is no clear standard as to which event should be used.

```
 1  contract MintableToken is StandardToken, Ownable {
 2  ...
 3      function mint(address _to, uint256 _amount) onlyOwner
            canMint public {
 4          totalSupply_ = totalSupply_.add(_amount);
 5          balances[_to] = balances[_to].add(_amount);
 6          Mint(_to, _amount);
 7          Transfer(address(0), _to, _amount);
 8      }
 9  ...
10  }
```

Fig. 12: Example of a simple mint function of a mintable ERC20 token.

The `burn` function in Figure 13 is not standard for ERC20 tokens because it allows the owner to burn tokens from any account. This can pose a security issue if the owner of the tokens is not a clearly defined and well tested smart contract. The burning of tokens works similarly to the minting of new tokens. After checking that the account the tokens should be burned from has enough of them in line 6, the total supply and the balance of the corresponding account are reduced in lines 8 and 7. Again two events are emitted in lines 9 and 10 to make sure that the action gets picked up by all token listeners.

Multi-signature operations As we discuss in Section 6.1, the operator contract is used as a multi-signature contract for the fund contract. It uses an array of different owners, split between internal owners and trust parties, to sign any transaction for the investment fund. Depending on the action a different amount of signatures might be needed.

```
contract FundToken is MintableToken {
...
    function burn(address _holder, uint256 _value) external onlyOwner
    {
        require(_holder != address(0));
        require(_value <= balances[_holder]);
        balances[_holder] = balances[_holder].sub(_value);
        totalSupply_ = totalSupply_.sub(_value);
        emit Burn(_holder, _value);
        emit Transfer(_holder, address(0), _value);
    }
...
}
```

Fig. 13: Burn function adapted for the ERCFund.

These signatures are created off-chain and are all sent within a single transaction which reduces transaction cost.

The function `verifyHotAction` in Figure 14 is used to check signatures for methods which only require signatures from internal owners. Within the code such an action is often referred to as a *hot action* as it can be performed on the fly. This function is called at the very beginning of functions which require the signatures of internal owners only, for example a price update. As an input in line 3 the three signature values v, r and s from the Elliptic Curve Digital Signature Algorithm (ECDSA) [51] are given which is the standard signature algorithm for the Ethereum blockchain and many others. Additionally a prehash value is given: This value is created in the caller function and is a hash of all relevant details of the execution of the function. We will see an example of this later on, but all prehashes follow the same format:

$$fund\ contract\ address + action\ identifier + function\ parameters + nonce$$

The fund contract address is simply the address where the fund contract is deployed at and is necessary to uniquely define the recipient of the signature. Every function executable via the operator class has a defined action identifier which is used to attribute the signature to a certain action. The function parameters include all necessary information needed for the execution, such as the new price of a share.
Lastly an important addition is a unique cryptographic nonce. Normally a nonce is a pseudo-random number created by a program or protocol to avoid external parties from guessing or calculating it and reducing possible attack vectors. In a deployed smart contract on the blockchain this is not possible because all variables of the contract are publicly visible, including the cryptographic nonce. Even though the nonce is publicly available it still serves an important purpose: It prevents the execution of replay attacks. A replay attack describes the attack of a third party maliciously executing a function with previous parameters repeatedly. This attack is highly relevant for any smart con-

```
contract FundOperator {
...
    function _verifyHotAction(uint8[] _sigV, bytes32[] _sigR,
        bytes32[] _sigS, bytes32 _preHash) view internal {
        require(_sigV.length >= hotThreshold);
        require(_sigR.length == _sigS.length && _sigR.length ==
            _sigV.length);
        bytes memory prefix = "\x19Ethereum Signed Message:\n32"
            ;
        bytes32 txHash = keccak256(prefix, _preHash);

        address lastAdd = 0;
        for (uint256 i = 0; i < hotThreshold; i++) {
            address recovered = ecrecover(txHash, _sigV[i],
                _sigR[i], _sigS[i]);
            require(recovered > lastAdd);
            require(isHotAccount[recovered]);
            lastAdd = recovered;
        }
    }
...
}
```

Fig. 14: Signature checking function of the Fund Operator class for a hot action.

tract using off-chain signatures, because once the transaction is published all signatures are visible to all participants of the blockchain. Without the addition of a nonce which increases after every successful transaction an attacker could reuse the signatures to disrupt the management of the investment fund.

Let us jump back to the code of the verifyHotAction function. Firstly, the input parameters are checked if they contain the appropriate amount of signatures in lines 4 and 5. The `hotThreshold` variable is set at contract creation and decides how many signatures are needed for a hot action. Secondly the prehash is transformed in line 7 to fit the Ethereum signature standard [65], for this a static prefix is prepended and the resulting string is hashed once more.

As the last step, starting with line 10, the function iterates over all given signatures and checks their correctness. The built-in `ecrecover` function takes as an input a message and a signature and recovers which address has signed the message to produce the given signature. Afterwards it is checked if this signature is actually an internal owner (also called hot account) of the operator contract. All signatures are checked in a loop which restricts the addresses to be strictly increasing. This is simply an efficient way to make sure that the signatures do not contain any duplicates (as every signer has to be unique).

Moving funds As an actively managed fund, one of the core features is the trading and moving of currency. We have already discussed that the investment fund can manage an arbitrary number of wallets which we call the wallet-layer. However most of the logic for handling many different wallets and different types of wallets happens in the operator-layer.

The operator smart contract saves a list of three different types of wallets: Hot wallets, trusted wallets and cold wallets. Hot wallets are internal wallets which are readily accessible and can be used at any time. It is possible to create transactions from them in a matter of seconds. Trusted wallets are external wallets which represent components with which the investment fund regularly interacts with and trusts. Most importantly exchanges fall in this category, but it can also include wallets to collect fees for example. Cold wallets require a special private key to unlock which is stored on a device which is not connected to the internet. How this works is covered in Section 6.2.

Two separate functions are included in the fund operator contract, the `requestEtherTransfer` and the `requestTokenTransfer` which are used to initiate either an Ether transfer or token transfer respectively. Their implementations are similar, here we take a look at how a token transfer is executed in Figure 15.

First of all, like in all methods, the signatures are checked for validity in line 5. For transfers a special method is used to implement the circle of trust feature covered in Section 6.2. The signature of this method has to include the type of token, the sender and the recipient in addition to the standard parameters. After verifying the signatures, the fund contract in Figure 16 is called to execute the transfer and emit an event. The token transfer method in the fund contract only acts as a middle-layer and directly relays the transfer to the wallet from which the tokens are supposed to be transferred from. Then in the wallet codein Figure 17, the token transfer is finally forwarded to

```
contract FundOperator {
...
    function requestTokenTransfer(uint8[] _sigV, bytes32[] _sigR
        , bytes32[] _sigS, ERC20 _token, FundWallet _from,
        address _to, uint256 _value) external hasFund {
        bytes32 preHash = keccak256(this, int256(Action.
            TokenTransfer), _token, _from, _to, _value, nonce);
        _verifyTransfer(_sigV, _sigR, _sigS, preHash, _from, _to
            , _value);

        fund.moveTokens(_token, _from, _to, _value);
        emit TokenTransferAuthorized(address(_token), _from, _to
            , _value);
    }
...
}
```

Fig. 15: Token transfer function of the fund operator contract.

```
contract Fund is FundWallet, Pausable {
...
    function moveTokens(ERC20 _token, FundWallet _from, address
        _to, uint256 _value)
        public
        onlyOwner
    {
        _from.sendTokens(_token, _to, _value);
        emit TokensMoved(address(_token), _from, _to, _value);
    }
...
}
```

Fig. 16: Token transfer function of the fund contract.

```
contract FundWallet is Ownable {
...
    function sendTokens(ERC20 _token, address _to, uint256 _value)
        public
        onlyOwnerOrInternal
        notNull(_to)
    {
        require(_value > 0);
        require(_token.transfer(_to, _value));
        emit TokensSent(_token, _to, _value);
    }
...
}
```

Fig. 17: Send tokens function of the wallet contract.

the token contract itself. It is checked if the token transfer is successful and then the execution completes.

Cold-wallet support Cold-wallet is a term for a wallet which is exclusively accessible by an address of which the private key was never stored on a device with internet connectivity. By keeping the private key completely separated from the internet, it makes it impossible for malicious hackers to steal it digitally. Most big exchanges keep a large percentage of their crypto-holdings on cold-wallets as a measure of security.
The implementation of cold-wallets happens strictly on the operator-layer. Cold-wallets are saved as a separate contract variable which maps the address of a cold wallet to the address which can unlock the wallet. This implementation is slightly different from other cold-wallets because the cold-wallet is a deployed smart contract. More commonly cold-wallets are simple accounts which means that the funds are stored by the address of which the private key is inaccessible. However, in order to unlock the smart contract a separate account is needed. This account is technically the one which is *cold* (never connected to the internet) while the smart contract holding the address is deployed live. We do not cover the code for cold-wallet support explicitly here as it is similar to the verifyTransfer function covered later in Figure 18, however all code is available on Github [60].
The difference between accessing cold-wallets and hot-wallets (private keys for these wallets are not separated from the internet) in the operator contract is rather small because the above described implementation for off-chain signatures makes it simple to use cold-keys. In addition to the usual signatures, a signature from the appropriate cold-key locking the cold-wallet is appended (which can be created on the isolated device). If this key matches the cold-wallet, the transaction is allowed to happen.

Circles of trust In Section 6.2 we have already partly discussed how to access internal wallets and how to send Ether or tokens from them. The circle of trust feature describes the separation of possible recipients of transfers and is covered in Section 3.5. Namely its purpose is to introduce a safe and trusted ecosystem with which the investment fund can interact at any time without having to worry about accidental transactions. Additionally it provides fund managers with the possibility to offer a layer of security for stakeholders because it can prevent the theft of funds.

This is possible because for certain actions in the fund, signatures from a list of third parties have to be provided. As mentioned before these can be e.g., large stakeholders. At first, all usual business partners and external wallets have to be explicitly added with the trust parties' consent. Only after these wallets are added as trusted can they be used in the day to day business of fund management. To understand why this is the case we have to take a look into the `verifyTransfer` method in Figure 18 which is called to verify every transfer of the investment fund.
In line 4 it is checked if the *from* wallet is a valid, internal wallet. For this it has to be either a hot or cold wallet. After checking that some value should be sent in line 5, the in Section 6.2 discussed verifyHotAction function is called. This function will run at some point during every possible action in the operator class. If the sender is a cold wallet, a separate segment starting in line 7 is entered where the last entry of the signatures is verified against the address of the cold wallet using a mapping securing all cold wallets with a dedicated cold key.
Lastly, it is checked in line 12 whether the recipient of the transaction is a trusted wallet, if this is not the case, trust party signatures are checked in addition to the usual signatures. The `verifyTrustPartyAction` method has the the same functionality as the verifyHotAction function, the signatures are simply matched against a different mapping. Also as with all other executions, the nonce is increased in line 15 to prohibit replayability.

```
contract FundOperator {
...
    function _verifyTransfer(uint8[] _sigV, bytes32[] _sigR,
        bytes32[] _sigS, bytes32 _preHash, FundWallet _from,
        address _to, uint256 _value) internal {
        require(isHotWallet[_from]  coldStorage[_from] != 0);
        require(_value > 0);
        _verifyHotAction(_sigV, _sigR, _sigS, _preHash);
        if (coldStorage[_from] != 0) {
            require(_sigR.length == _sigS.length && _sigR.length
                == _sigV.length);
            uint256 coldKeyPos = _sigV.length - 1;
            _verifyColdStorageAccess(_sigV[coldKeyPos], _sigR[
                coldKeyPos], _sigS[coldKeyPos], _preHash, _from)
                ;
        }
        if (!isTrustedWallet[_to]) {
            _verifyTrustPartyAction(_sigV, _sigR, _sigS,
                _preHash);
        }
        nonce = nonce.add(1);
    }
...
}
```

Fig. 18: Function of the operator contract used to verify token and Ether transfers.

7 Conclusion

Blockchain technology has enabled many traditional fields to utilize it for new, unique use cases, the investment field is no exception to this. The last two years have brought a lot of attention to the possibility of investing in cryptocurrency, but brought only few services which help non-technical customers invest easily. In this paper we present a novel semantics based on the Ethereum network with a strong focus on smart contract interactions. Furthermore we developed a fully-fledged investment fund software built entirely on the Ethereum blockchain using smart contracts and provide motivation on how our semantics could ensure security properties thereof.

The semantics is based on a paper by Grishchenko et al. [48] and represents the blockchain in terms of global state information. However, changes in the global state are represented using a big-step semantics which corresponds to the high-level execution of a transaction and does not consider execution on an EVM bytecode level. Smart contracts are defined by clearly separating immutable and mutable information. They consist of fixed behavior for all functions, a set of contract variables (persistent storage) and their balance. Additionally the semantics allows for the execution of a smart contract function to execute another transaction (i.e., an internal transaction) which makes the semantics suitable for representing complex executions ranging over multiple different smart contracts. We use the semantics to model common smart contract designs, such as multi-signature wallets and tokens, and demonstrate how their functions can be translated into the semantics. Finally, the multi-signature wallet is used to formally verify a desired property of the investment fund which can prevent currency from being stolen.

In the second part of the thesis the implementation of the investment fund *ERCFund* [60] is covered. ERCFund makes it possible to invest into an actively managed portfolio of ERC20-Tokens and Ether by introducing an on-demand minted and burned token as the medium for shares in the fund. Moreover, customers can buy shares/tokens at any time, the fund offers cold-wallet support and has a significantly lower gas cost than traditional multi-signature wallets by utilizing off-chain signatures. Design choices are discussed in detail and justified based on issues like blockchain related attack vectors and current market trends. The paper offers an in-depth coverage of the most crucial parts of the smart contracts and gives insight into state-of-the-art smart contract design.

While the blockchain sector is still young, we believe it is essential to make first steps to allow traditional financial institutions to adapt to this new environment and consequently accelerate growth. Currently a big deterrent is the volatility and lacking security features of blockchains and smart contracts [67], [39] which marks blockchain itself still as a venturesome investment field. This paper works towards solving these issues by providing a user-centric semantics for smart contract verification and a state-of-the-art, easily usable and secure investment fund software.

References

1. Bitcoin block explorer - blockchain. https://blockchain.info/. Accessed: 2018-04-25.
2. Bitcoin energy consumption index. https://digiconomist.net/bitcoin-energy-consumption. Accessed: 2018-04-25.
3. Bitcoin hashrate distribution. https://blockchain.info/pools. Accessed: 2018-04-26.
4. Coinist 50 biggest icos. https://www.coinist.io/biggest-icos-chart/. Accessed: 2018-05-04.
5. Cost of a 51% attack. https://gobitcoin.io/tools/cost-51-attack/. Accessed: 2018-04-26.
6. Crypto20: The first tokenized cryptocurrency index fund - fund value. https://crypto20.com/en/. Accessed: 2018-05-07.
7. Cryptocurrency market capitalizations — coinmarketcap. https://coinmarketcap.com/. Accessed: 2018-05-07.
8. Double-spending types of attacks. https://en.bitcoinwiki.org/wiki/Double-spending#Types_of_attacks. Accessed: 2018-05-01.
9. Ethereum multisignature wallet. https://github.com/gnosis/MultiSigWallet. Accessed: 2018-05-10.
10. Ethereum problems. https://github.com/ethereum/wiki/wiki/Problems. Accessed: 2018-05-01.
11. Etherscan: The ethereum block explorer. etherscan.io. Accessed: 2018-03-26.
12. Etherscan token tracker. https://etherscan.io/tokens. Accessed: 2018-05-07.
13. Gbtc - bitcoin investment trust - invest in bitcoin with grayscale. https://grayscale.co/bitcoin-investment-trust/#market-performance. Accessed: 2018-05-05.
14. Ico drops - ended ico - bancor. https://icodrops.com/bancor/. Accessed: 2018-05-04.
15. Ico drops - ended ico - filecoin. https://icodrops.com/filecoin/. Accessed: 2018-05-04.
16. Multi-sig wallet contract code. Ethereum contract at address 0x0af5f9a338870073707939b39A0ea3eBC0bAD00b. Source code visible at https://etherscan.io/address/0x0af5f9a338870073707939b39a0ea3ebc0bad00b#code. Code of early version of Parity's multisig-wallet including major bug. Accessed: 2018-05-03.
17. Neo white paper. http://docs.neo.org/en-us/. Accessed: 2018-01-14.
18. Neo white paper. https://github.com/neo-project/docs/blob/master/en-us/index.md. Accessed: 2018-04-26.
19. Notice of exempt offering of securities, issuer: Ton issuer inc. https://www.sec.gov/Archives/edgar/data/1729650/000095017218000030/xslFormDX01/primary_doc.xml. First Notice of TON Investment, Accessed: 2018-04-05.
20. Notice of exempt offering of securities, issuer: Ton issuer inc. https://www.sec.gov/Archives/edgar/data/1729650/000095017218000060/xslFormDX01/primary_doc.xml. Second Notice of TON Investment, Accessed: 2018-04-05.
21. Official go implementation of the ethereum protocol. https://github.com/ethereum/go-ethereum. Accessed: 2018-04-19.
22. On the parity multi-sig wallet attack. https://medium.com/blockcat/on-the-parity-multi-sig-wallet-attack-83fb5e7f4b8c. Accessed: 2018-05-03.
23. Openzeppelin - standard erc20 token. https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/StandardToken.sol. Accessed: 2018-05-05.

24. Openzeppelin safemath. https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol. Accessed: 2018-05-02.
25. Oraclize - blockchain oracle service, enabling data-rich smart contracts. http://www.oraclize.it/. Accessed: 2018-05-05.
26. Secure bitcoin storage - coinbase. https://www.coinbase.com/security?locale=en. Accessed: 2018-05-10.
27. Soldity v0.4.11 common patterns withdrawal from contracts. https://solidity.readthedocs.io/en/v0.4.11/common-patterns.html#withdrawal-from-contracts. Accessed: 2018-05-02.
28. Solidity version 0.4.23. https://solidity.readthedocs.io/en/v0.4.23/. Accessed: 2018-04-29.
29. Total market capizalization. https://coinmarketcap.com/charts/. Accessed: 2018-04-29.
30. What are atomic swaps? https://www.cryptocompare.com/coins/guides/what-are-atomic-swaps/. Accessed: 2018-05-05.
31. What is the raiden network? https://raiden.network/101.html. Accessed: 2018-07-24.
32. civic whitepaper. https://tokensale.civic.com/CivicTokenSaleWhitePaper.pdf, 2017. Accessed: 2018-07-24.
33. Gavin Andresen. Bitcoin improvement proposal: 50. https://github.com/bitcoin/bips/blob/master/bip-0050.mediawiki, 2013. Accessed: 2018-05-01.
34. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, A Rastogi, T Sibut-Pinote, N Swamy, and S Zanella-Beguelin. Formal verification of smart contracts. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security-PLAS16*, pages 91–96, 2016.
35. John Biggs. Exit scammers run off with $ 660 million in ico earnings. https://techcrunch.com/2018/04/13/exit-scammers-run-off-with-660-million-in-ico-earnings/. Accessed: 2018-05-08.
36. Vitalik Buterin. Comment on intial minimum stake for proof of stake protocol. https://www.reddit.com/r/ethereum/comments/6tj5d0/any_updates_on_ethereums_pos/dllfyd1/. Accessed: 2018-04-26.
37. Vitalik Buterin. Ethereum 2.0 mauve paper. https://docs.google.com/document/d/1maFT3cpHvwn29gLvtY4WcQiI6kRbN_nbCf3JlgR3m_8/edit. Accessed: 2018-04-26.
38. Vitalik Buterin. Hard fork completed. https://blog.ethereum.org/2016/07/20/hard-fork-completed/. Accessed: 2018-05-03.
39. Vitalik Buterin. Security alert [11/24/2016]: Consensus bug in geth v1.4.19 and v1.5.2. https://blog.ethereum.org/2016/11/25/security-alert-11242016-consensus-bug-geth-v1-4-19-v1-5-2/, 2016. Accessed: 2018-05-01.
40. Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.
41. Dimitri Chupryna, Maksym Muratov, Konstantin Pysarenko, and Ruslan Gavrylyuk. Taas: Token-as-a-service. https://bravenewcoin.com/assets/Whitepapers/TaaS-whitepaper.pdf, 2017. Accessed: 2017-10-18.
42. CodeTract. Inside an ethereum transaction. https://medium.com/@codetractio/inside-an-ethereum-transaction-fa94ffca912f, 2017. Accessed: 2018-04-25.

43. Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*, pages 79–94. Springer, 2016.
44. ”devops199”. anyone can kill your contract. `https://github.com/paritytech/parity/issues/6995`. Accessed: 2018-05-03.
45. John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.
46. Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. Technical report, 2015.
47. Victor Fleischer. Two and twenty: Taxing partnership profits in private equity funds. *NYUL Rev.*, 83:1, 2008.
48. Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. *arXiv preprint arXiv:1802.08660*, 2018.
49. 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.
50. Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. Kevm: A complete semantics of the ethereum virtual machine. Technical report, 2017.
51. Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.
52. Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. NDSS, 2018.
53. Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. `https://eprint.iacr.org/2016/889.pdf`, 2017. Accessed: 2018-03-29.
54. Felix Lau, Stuart H Rubin, Michael H Smith, and Ljiljana Trajkovic. Distributed denial of service attacks. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 3, pages 2275–2280. IEEE, 2000.
55. 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.
56. Maker. The dai stablecoin system. `https://makerdao.com/whitepaper/`. Accessed: 2018-04-29.
57. Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.
58. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
59. OpenZeppelin. Simple savings wallet. `https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/examples/SimpleSavingsWallet.sol`. Accessed: 2018-04-29.
60. Jakob Schneider. Ercfund an open-ended investment fund implementation on the ethereum blockchain for managing erc20 tokens. `https://github.com/ScJa/ercfund`. Accessed: 2018-05-24.
61. Daniel Schwartzkopff, Schwartzkopff Luke, Raymond Botha, Matthew Finlayson, and Frans Cronje. Crypto20: The first tokenized cryptocurrency index fund. `https://static.crypto20.com/pdf/c20-whitepaper.pdf`, 2017. Accessed: 2018-01-14.
62. SECG SEC. 2: Recommended elliptic curve domain parameters. *Standards for Efficient Cryptography Group, Certicom Corp*, 2000.

63. Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level language. *arXiv preprint arXiv:1801.00687*, 2018.
64. Andrei Stefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. In *ACM SIGPLAN Notices*, volume 51, pages 74–91. ACM, 2016.
65. Martin Holst (@holiman) Swende. Erc: Signed data standard. `https://github.com/ethereum/EIPs/issues/191`, 2016. Accessed: 2018-04-12.
66. Reto Trinkler and Mona El Isa. Melon protocol: A blockchain protocol for digital asset management. `https://github.com/melonproject/paper/blob/master/melonprotocol.pdf`. Accessed: 2018-07-07.
67. Peter Vessenes. Deconstructing theDAO attack: A brief code tour. `http://vessenes.com/deconstructing-thedao-attack-a-brief-code-tour/`, 2016. Accessed: 2018-01-14.
68. Fabian Vogelsteller and Vitalik Buterin. Erc-20 token standard. `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md`. Accessed: 2018-05-05.
69. Will Warren and Amir Bandeali. 0x: An open protocol for decentralized exchange on the ethereum blockchain. `https://github.com/0xProject/whitepaper/blob/master/0x_white_paper.pdf`, 2017. Accessed: 2018-04-26.
70. Tim M. Zagar, Jani Valjavec, Zenel Batagelj, Ervin U. Kovac, and Ales Lekse. Iconomi - open fund management plagform to disrup the investment industry. https://coss.io/documents/white-papers/iconomi.pdf. Accessed:2018-07-07.
71. Vlad Zamfir. Introducing casper 'the friendly ghost'. *Ethereum Blog URL: `https://blog.ethereum.org/2015/08/01/introducing-casper-friendly-ghost`*, 2015.