

Semantic Verification of Ethereum Smart Contracts using KEVM

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Artificial Intelligence

by

Sascha Pleßberger, BSc.

Registration Number 01425186

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.Ing. Dr.techn. Gernot Salzer

Assistance: Ass.Prof.in Dipl.-Ing.in Mag.a rer.soc.oec. Dr.in techn. Monika di Angelo

Vienna, February 14, 2026

Sascha Pleßberger

Gernot Salzer

Declaration of Authorship

Sascha Pleßberger, BSc.

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

I further declare that I have used generative AI tools only as an aid, and that my own intellectual and creative efforts predominate in this work. In the appendix "Overview of Generative AI Tools Used" I have listed all generative AI tools that were used in the creation of this work, and indicated where in the work they were used. If whole passages of text were used without substantial changes, I have indicated the input (prompts) I formulated and the IT application used with its product name and version number/date.

Vienna, February 14, 2026

Sascha Pleßberger

Acknowledgements

I would like to express my sincere gratitude to my advisor, Gernot Salzer, for his continuous support, guidance, and patience throughout this thesis. His encouragement and feedback were instrumental in bringing this work to completion.

I am especially thankful to my partner, Corinna Wolfinger, for believing in me and for motivating me to push on, particularly during the more challenging phases of this project.

Finally, I would like to thank my family for their constant support and encouragement over the years.

Abstract

Ethereum smart contracts often hold a significant amount of funds. Since they are virtually immutable and are often exposed to malicious agents incentivised by monetary gain, semantic correctness is a critical security concern. Established testing techniques are often insufficient to guarantee correctness across all possible execution paths. As such, formal verification is an essential tool in strengthening such security guarantees. This thesis investigates symbolic-execution-based verification of Ethereum smart contracts using the KEVM framework and two higher-level tools built on top of it: ACT and Kontrol.

Our work considers questions regarding proof expressiveness and construction, usability and understandability of both proof definitions and generated proof artefacts. We will examine if and what practical challenges occur during the use of KEVM and its related tools, including syntax, available debuggers and proof or refutation analysis. Next, an evaluation is done on how semantic properties can be specified across all tools and how accurate they are, highlighting the trade-offs between abstraction levels.

Furthermore, we verify semantic properties of ERC20 token contracts, with a focus on determining whether certain reports in the Common Vulnerabilities and Exposures (CVE) database are indeed correct or can be refuted using KEVM. To that end, we examine the vulnerability reported, give a formal argument against the reported violation and from there construct a proof using Kontrol. Using this approach, we demonstrate how semantic properties can be expressed and verified within the framework.

Finally, we analyse community activity around KEVM and its ecosystem. GitHub repository metrics and Discord communication data are evaluated to assess development activity, contributor dynamics, and user support patterns. This combined technical and empirical perspective provides a comprehensive view of KEVM as both the formal verification framework and the developer ecosystem.

Kurzfassung

Ethereum-Smart Contracts verwalten häufig erhebliche finanzielle Werte. Da sie praktisch unveränderlich sind und häufig böswilligen Akteuren ausgesetzt sind, die durch finanziellen Gewinn motiviert sind, stellt die semantische Korrektheit eine zentrale Sicherheitsanforderung dar. Etablierte Testmethoden reichen oft nicht aus, um die Korrektheit über alle möglichen Ausführungspfade hinweg zu gewährleisten. Daher stellt die formale Verifikation ein wesentliches Mittel dar, um solche Sicherheitsgarantien zu stärken. Diese Arbeit untersucht die auf symbolischer Ausführung basierende Verifikation von Ethereum-Smart-Contracts unter Verwendung des KEVM-Frameworks sowie zweier darauf aufbauender Werkzeuge auf höherer Abstraktionsebene: ACT und Kontrol.

Diese Arbeit behandelt Fragestellungen hinsichtlich der Ausdrucksstärke und Konstruktion von Beweisen sowie der Nutzbarkeit und Interpretierbarkeit sowohl von Beweisdefinitionen als auch von generierten Beweisartefakten. Es wird untersucht, ob und welche praktischen Herausforderungen bei der Verwendung von KEVM und zugehörigen Werkzeugen auftreten, einschließlich der Syntax, der verfügbaren Debugging-Werkzeuge sowie der Analyse von Beweisen und Gegenbeweisen. Anschließend erfolgt eine Evaluierung, wie semantische Eigenschaften über alle Werkzeuge hinweg spezifiziert werden können und wie präzise diese spezifiziert werden, wobei insbesondere die Zielkonflikte zwischen unterschiedlichen Abstraktionsebenen hervorgehoben werden.

Darüber hinaus verifizieren wir semantische Eigenschaften von ERC20-Token-Smart-Contracts mit besonderem Fokus darauf, ob bestimmte Einträge in der Common Vulnerabilities and Exposures (CVE)-Datenbank tatsächlich korrekt sind oder mithilfe von KEVM widerlegt werden können. Zu diesem Zweck analysieren wir die gemeldete Schwachstelle, formulieren ein formales Argument gegen die behauptete Verletzung und konstruieren darauf aufbauend einen Beweis unter Verwendung von Kontrol. Dabei zeigen wir, wie semantische Eigenschaften innerhalb des Frameworks formuliert und verifiziert werden können.

Abschließend untersuchen wir die Community-Aktivität rund um KEVM und dessen Ökosystem. Dazu werden GitHub-Repository-Metriken sowie Kommunikationsdaten aus Discord ausgewertet, um Entwicklungsaktivität, Dynamiken der Beitragenden sowie Muster im Nutzer-Support zu analysieren. Diese kombinierte Perspektive aus technischer und empirischer Sicht liefert eine ganzheitliche Betrachtung von KEVM sowohl als formales Verifikationsframework als auch als Entwickler-Ökosystem.

Contents

| | |
|--|-----------|
| Abstract | iv |
| Kurzfassung | v |
| Contents | vi |
| 1 Introduction | 1 |
| 1.1 Motivation | 1 |
| 1.2 Problem Statement | 2 |
| 1.3 State of the Art | 3 |
| 1.4 Structure of this Thesis | 3 |
| 2 Background | 5 |
| 2.1 Ethereum | 5 |
| 2.2 EVM | 5 |
| 2.3 The Need for Formal Verification | 6 |
| 2.4 The K Framework and KEVM | 6 |
| 2.5 klab and ACT | 8 |
| 2.6 Kontrol | 8 |
| 3 Technical Background | 10 |
| 3.1 KEVM | 10 |
| 3.2 ACT | 20 |
| 3.3 Kontrol | 24 |
| 4 Verification using Symbolic Execution in KEVM | 30 |
| 4.1 CVE Reports | 30 |
| 4.2 Formal Argument | 35 |
| 4.3 KEVM | 39 |
| 4.4 ACT | 42 |
| 4.5 Kontrol | 43 |
| 5 Evaluation | 50 |
| 5.1 User Experience | 50 |

| | | |
|----------|--|-----------|
| 5.2 | General Tool Results Analysis | 57 |
| 5.3 | Community Support | 61 |
| 6 | Conclusion | 74 |
| | Overview of Generative AI Tools Used | 77 |
| | Text and Grammar | 77 |
| | Help with Python for Visualisation Tasks | 77 |
| | Overall Prompt Summary | 81 |
| | List of Figures | 83 |
| | List of Tables | 84 |
| | Bibliography | 85 |
| | Addendum | 87 |
| | A.1 Smart Contract Code | 87 |
| | A.2 KEVM Test Specifications | 96 |
| | A.3 Kontrol Test Specifications | 105 |

Introduction

1.1 Motivation

Ethereum's smart contracts have become increasingly popular within the cryptocurrency community over the last few years. They are pieces of code that are deployed to the Ethereum blockchain. There, they act as tamper-evident agents on the network that, upon instruction, automatically perform a specified set of actions determined by their deployed bytecode. As such, they have been widely used in consensus-driven applications in the cryptocurrency space. However, coding faults may enable malicious actors to weaken the trust placed in these smart contracts, or, worse, grant them direct access to funds held within the contract. Furthermore, it is often not possible to fix these issues, since we cannot directly modify the code of a deployed smart contract.

With the rise of cryptocurrencies and digital assets, as well as smart contracts that operate on and with such assets, the financial damage caused when flaws in these systems are discovered and exploited has increased. According to the REKT Database by DEFIYIELD¹, as of 31st October 2022, nearly 62 billion US Dollars worth of digital assets have been lost in hacks and other exploits.

All of this highlights the need for enhanced security when developing and deploying code in this space. Specifically, a sound, formal semantic verification tool would allow us to give strong guarantees regarding the security aspects of such code. Smart contracts, in particular, lend themselves well to this approach, as they are typically small programs. Although they may be small, the context in which they are executed is anything but. The Ethereum blockchain is a large network, and there are many pitfalls and peculiarities to consider when coding. Smart contracts are compiled and deployed in what is known as EVM (Ethereum Virtual Machine) bytecode. In conclusion, all of this means that once this bytecode is made public (deployed) to the blockchain, function calls to the

¹<https://defiyield.app/rekt-database>

smart contract are executed within the Ethereum Virtual Machine and deterministically influence the global state of the Ethereum blockchain.

The semantic framework \mathbb{K} , reviewed in [Rosu, 2017], and its extension KEVM, defined by [Hildenbrandt et al., 2018], provides such fully executable semantics for Ethereum smart contracts and consists of Ethereum semantics formalised in \mathbb{K} , a fully compliant reference interpreter and a smart contract program verifier.

1.2 Problem Statement

Many semantic program verification tools are developed independently by individual researchers. As a result, they are commonly abandoned shortly after the initial research goal is achieved and may never achieve sustained adoption among the developer community. Furthermore, these tools are often complex to operate and require specialised knowledge to use effectively.

The \mathbb{K} semantic framework has attracted a consistent and active developer base, as evidenced by the \mathbb{K} Semantic Framework’s GitHub repository², owing to the aforementioned advantages of the tool. Even its implementation of the EVM bytecode language, KEVM, has a stable, albeit smaller, and active developer and user base, according to the KEVM GitHub repository³. Furthermore, there are tools being introduced that allow for the definition of semantic properties on a higher level that are then compiled to KEVM specifications and proofs, such as ACT⁴ and integrations into the Foundry Forge testing framework⁵, now re-branded as Kontrol⁶. This makes writing semantic property specifications and proofs more accessible. As such, it seems to be a very promising choice as a formal verification tool for Ethereum smart contracts. The thesis aims to investigate the use of KEVM’s program verification capabilities and to simultaneously verify the semantic correctness of selected smart contracts.

Note that KEVM was developed to verify smart contracts on the semantics of the Ethereum EVM. But more often than not, errors and vulnerabilities result from the implementation not behaving as intended with respect to the semantics of a smart contract’s code. We want to observe how the tool performs when we extend the semantic rules to test the correctness of smart contracts that are required to have certain semantic properties, for example, smart contracts that follow the ERC20 token standard.

To specify, the thesis will aim at answering the following research questions:

1. As an individual with specialised education in the field, what difficulties arise when working with the verifier?

²<https://github.com/runtimeverification/k>

³<https://github.com/runtimeverification/evm-semantics>

⁴<https://github.com/ethereum/act>

⁵<https://github.com/foundry-rs/foundry>

⁶<https://github.com/runtimeverification/kontrol>

2. To what extent is it possible to define accurate rules for smart contracts' semantic properties in KEVM, ACT and Kontrol?
3. How can we prove semantic properties and the absence of certain vulnerabilities, specifically concerning the ERC20 token standard? Specific examples include arithmetical exceptions like overflows and underflows, (in)ability to send ether to and from a contract, etc.
4. Given a known semantic property and a smart contract in violation of such a property, what can we deduce other than the violation itself?

1.3 State of the Art

We can broadly categorise verification tools by their approach: either by actively seeking a vulnerability or by proving its non-existence, typically by establishing a security property which rules out the vulnerability. KEVM belongs to the latter category, and as such, we are focusing on this category in this thesis. Nonetheless, for a complete overview, we can recommend the review by [Rameder et al., 2022].

EthIsabelle, discussed in [Hirai, 2017], formalised the semantics of the Ethereum EVM in the theorem prover language Lem and provided some proofs for certain smart contract safety properties in Isabelle/HOL, an interactive theorem prover. Based on this project, [Amani et al., 2018] extended the formalisation with a sound, bytecode-level program logic, enabling semi-automatic proofs of smart contract properties.

[Grishchenko et al., 2018] considers EtherTrust the first fully sound static analysis tool. It focuses mainly on the so-called Single-Entrancy security property, which provides a security guarantee against the popular reentrancy exploit. Its successor, eThor, was introduced by [Schneidewind et al., 2020] and features a newly developed high-level specification of Horn-clause-based static analyses, HoRST, which has been shown to be sound.

1.4 Structure of this Thesis

This thesis is structured to gradually introduce the theoretical background, technical tools, and practical verification workflow for the semantic verification of Ethereum smart contracts using the KVM and related tools.

Chapter 1 introduced our motivation for this thesis by highlighting the need for formal verification in the context of blockchain systems, outlined our problem statement and the research questions guiding this thesis, and, lastly, discussed the current state of the art in smart contract verification.

Chapter 2 provides the necessary background on Ethereum and the Ethereum Virtual Machine (EVM). It introduces the execution model of smart contracts and discusses common security risks, motivating the need for formal verification techniques. The

chapter further introduces the K semantic framework and the KEVM project, along with higher-level tools such as ACT and Kontrol, which extend the usability of KEVM for practical verification workflows.

Chapter 3 outlines the technical foundations required to understand and apply symbolic execution in KEVM, ACT, and Kontrol. It explains how verification specifications are constructed for each, displaying their respective strengths and weaknesses with respect to semantic expressiveness.

Chapter 4 demonstrates applications of symbolic execution using KEVM and related tools to verify semantic properties of real-world smart contracts. We focus on ERC20 token implementations and selected vulnerability reports from the Common Vulnerabilities and Exposures (CVE) database. Furthermore, this chapter shows how semantic arguments can be translated into formal verification proofs and how incorrect vulnerability claims can be formally refuted.

Chapter 5 evaluates the effectiveness of KEVM, ACT, and Kontrol. We will highlight some considerations relevant to tool usability, expressiveness and support, among other factors. Additionally, this chapter analyses the KEVM ecosystem by inspecting GitHub repository metrics and community communication data. This provides a broader perspective on tool maturity and adoption.

Chapter 6 concludes the thesis by summarising the key findings, discussing limitations, and future research directions, including possible improvements to verification workflows and opportunities for wider adoption of formal verification in smart contract development.

Background

2.1 Ethereum

Ethereum, introduced in 2013 by Vitalik Buterin in [Buterin et al., 2014] and released in 2015, is a decentralised blockchain that spearheaded the evolution of blockchain technology. In contrast to Bitcoin, which primarily functions as a peer-to-peer electronic cash system, Ethereum is programmable and enables the development and deployment of decentralised applications. At the core of this functionality is the Ethereum Virtual Machine (EVM). It is a quasi-Turing-complete, stack-based virtual machine that executes bytecode deterministically, as stated by [Wood et al., 2014]. This bytecode is at the heart of decentralised apps (dApps) and smart contracts, which are semi-self-executing agents that represent agreements with the terms written in bytecode. Note that these are not fully autonomous, as execution still needs to be initialised via a call from a personal account. Furthermore, smart contracts are typically written in Solidity or Vyper and then compiled into EVM bytecode.

2.2 EVM

The Ethereum Yellow Paper [Wood et al., 2014] defines the Ethereum Virtual Machine (EVM) as a global state machine that processes transactions and executes smart contracts. Each transaction deterministically transforms the global state of the blockchain, and every node in the Ethereum network maintains its own instance of the EVM in order to independently verify state transitions. Consistency across the decentralised network is ensured through a consensus protocol, which guarantees that all honest nodes eventually agree on the same sequence of state updates.

Internally, the EVM is a stack-based virtual machine that operates on a last-in, first-out (LIFO) stack with a maximum depth of 1024 elements. This stack serves as the primary

mechanism for passing operands to arithmetic and logical instructions. In addition to the stack, the EVM maintains both volatile memory and persistent storage. Memory is a byte-addressable space that exists only for the duration of a single transaction and is reset after execution, while storage represents a long-term contract state and is implemented as part of a Merkle Patricia Trie. This structure encodes the complete set of accounts, balances, and smart contracts in a deterministic and cryptographically verifiable form, which serves as the robust global state of the blockchain.

To prevent the execution of long-running or non-halting code, the EVM employs the gas mechanism, in which each opcode is assigned a specific gas cost. During execution, gas is consumed proportionally to the computational cost and the storage resources required by each operation. If the available gas is exhausted, execution is aborted, and state changes are reverted. This ensures that programs cannot run indefinitely and provides incentives to minimise execution cost and storage consumption, hence EVM is quasi-Turing-complete.

2.3 The Need for Formal Verification

Although Ethereum’s smart contract can be extremely powerful and versatile, they are still prone to security vulnerabilities (e.g. reentrancy, integer over- & underflows) due to irreversible execution and adversarial conditions, as highlighted by [Atzei et al., 2017]. Unit tests and similar standard testing methods are insufficient when correctness of execution is a requirement, as they fail to prove correctness in every possible execution path. The formal verification method of symbolic execution, however, is able generate such proofs. Symbolic execution achieves this, as outlined in [Luu et al., 2016], via abstracting inputs as symbolic variables instead of concrete values and then exploring all feasible execution paths by solving constraints via Satisfiability Modulo Theories (SMT) solvers (e.g., Z3).

In this thesis, we will investigate symbolic-execution-based verification using KEVM and other tools that build upon the KEVM framework.

2.4 The K Framework and KEVM

\mathbb{K} , as defined in [Rosu, 2017], is a rewrite-based, semantic framework designed for formally specifying programming languages and virtual machines. Unlike many established approaches, which describe semantics in informal pseudocode, \mathbb{K} uses a precise mathematical foundation in which language definitions are executable, modular, and human-readable. Executable means that code can be run directly in \mathbb{K} for testing and verification, modularity ensures that the language semantics can be incrementally extended, and human-readability aims to maintain an intuitive notation.

This framework has been successfully applied to formally specify real-world systems, including C, Java, and, in particular, the Ethereum Virtual Machine by [Hildenbrandt et al., 2018]

through **KEVM**.

2.4.1 The \mathbb{K} Framework

The \mathbb{K} framework is structured around three core components that define both the operational behaviour of programs and the logical foundations to reason about them. First is the configuration structure, which represents program states as a nested collection of cells within the framework. These cells correspond to different parts of the execution state, such as memory, stack, environment, and storage, and together form a symbolic snapshot of the program at a given point in execution. This structured representation allows program states to be manipulated and inspected in a formal way.

Program behaviour in \mathbb{K} is defined through rewrite rules, which specify state transitions. Each rule describes a pattern that matches part of the current configuration and replaces it with a new configuration, thus modelling a single computational step. In the context of KEVM, individual EVM opcodes are specified as rewrite rules that transform the execution state according to their semantics.

Finally, \mathbb{K} employs matching logic, providing a formal reasoning layer on top of executable semantics. Matching logic allows configurations to be interpreted as logical formulas, enabling program properties to be expressed and proven directly over the operational model. This unification of execution and logic is a central feature of \mathbb{K} , as it allows the same specification to serve both as a simulator and as a basis for formal verification.

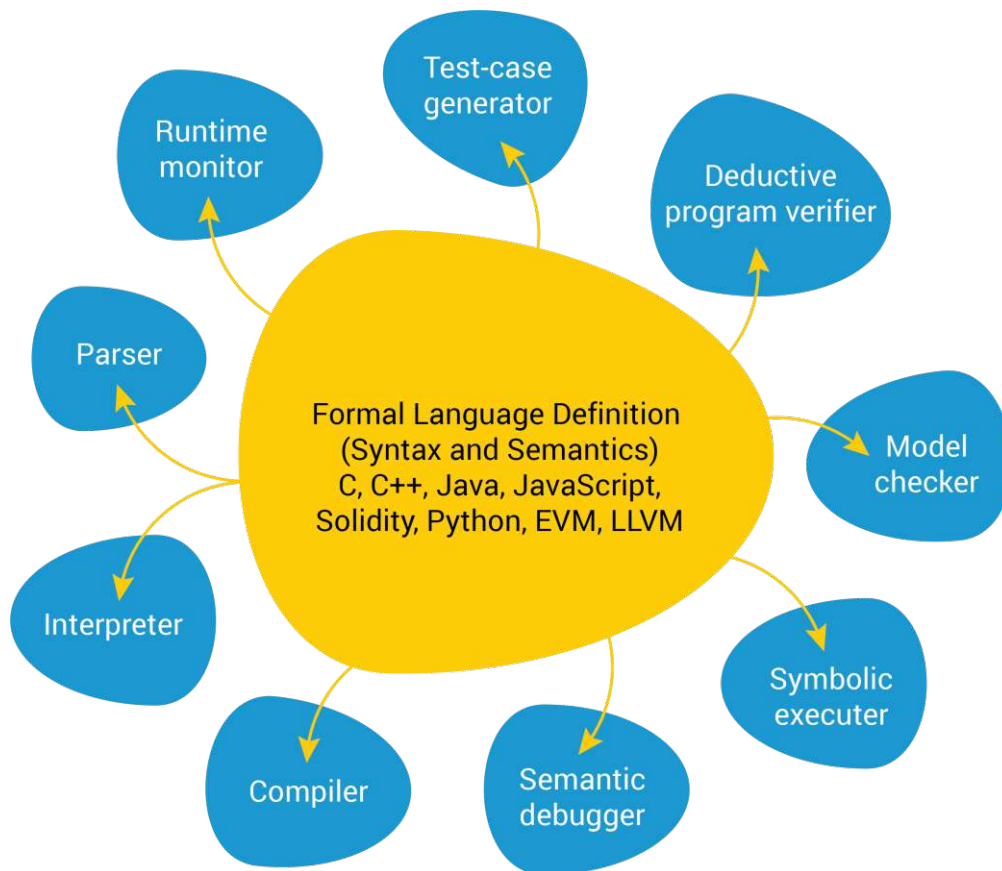
2.4.2 KEVM Implementation

The KEVM project implements the complete Ethereum Virtual Machine specification within the \mathbb{K} framework, providing a fully executable and formally defined model of EVM behaviour. All opcodes are specified with operational semantics, including gas cost calculations, allowing program execution to be simulated and reasoned about at the same level as in the real EVM.

Apart from individual instructions, KEVM captures the more extensive execution model of Ethereum, including memory management, persistent storage, and the interaction between stack, memory, and storage. Exceptions, such as out-of-gas conditions, stack underflows, and other runtime errors, are defined in the semantic model, enabling formal reasoning not only about successful executions but also about failure modes.

Defining EVM semantics in \mathbb{K} has certain advantages. First, KEVM's tools and executions behave verifiably correctly, as the specifications have been tested against Ethereum's official test suite. Furthermore, the \mathbb{K} framework can generate a parser, interpreter and many other tools, as depicted in Figure 2.1, if provided with the correct syntax and semantics definitions. Lastly, given how languages are defined and how \mathbb{K} operates, we can create strong, formal proofs for EVM properties.

¹<https://runtimeverification.com/blog/k-framework-an-overview>

Figure 2.1: The \mathbb{K} Framework¹

2.5 klab and ACT

ACT was introduced in [DappHub, 2019] as the specification language for the klab tool suite to generate and debug proofs with KEVM. Using ACT, it was possible to write more concise proof constraints, allowing users to adopt an easier process for proof generation. The klab project seems to have been largely discontinued, although ACT has been kept alive as a specification language to write proofs for three supported backends: SMT, Coq and HEVM.

2.6 Kontrol

Introduced as an extension to KEVM, Kontrol has evolved into a formal verification tool that combines the usability of the modern developer framework Foundry with the semantic rigour of KEVM. By integrating with the Foundry toolchain, Kontrol enables specifications to be written directly in Solidity, allowing developers to reuse existing test suites and verification workflows. This design decision significantly lowers the barrier to

entry for formal verification.

Kontrol is fully open source under the BSD-3 Clause License, allowing unrestricted inspection, modification, and reuse. The tool is intended to scale to realistic smart contracts by supporting automated verification against formal specifications, along with proof optimisations based on lemmas, loop invariants, and bounded model checking. Its verification process remains based on KEVM, preserving the formal correctness guarantees of the underlying semantics. In addition, Kontrol is able to integrate into continuous integration (CI) pipelines, supporting automated verification as part of standard development workflows and aiding in the adoption of formal methods in practical software engineering settings.

Technical Background

We will now further examine how testing and, to that end, verification of smart contracts in an EVM environment can be achieved using KEVM's symbolic execution branch. We will begin with the initial, low-level approach by writing and compiling verification tests directly in \mathbb{K} (EVM) and work our way through some more modern approaches like ACT and Kontrol, the Foundry integrated KEVM test environment.

3.1 KEVM

Writing tests in KEVM will require us to look at how \mathbb{K} (EVM) defines its requirements for specifications.

3.1.1 Syntax and Semantics of data structures

[Hildenbrandt et al., 2018] defines a K specification through the following three main components:

1. Syntax definition in EBNF form
2. state/configuration description
3. transitional rules to steer program execution

Let us start with an introduction to some simple syntax definitions. For positive integers, `uint256` is a commonly used datatype. Integer safety plays an important role in smart contracts. Let us define some bounds as an example.

We introduce two new terms, using the `syntax` keyword, type definition `Int` and additional attribute `function`. We will therefore have two functions without input

terms, i.e. constants, or *pure* functions in the smart contract context, which return some constant integer. Note that many operators in \mathbb{K} , and by extension KEVM, like Int^{\wedge} are infix and, for the sake of brevity “work as expected”, such as is the case here with integer exponentiation.

```

1 syntax Int ::= "uint256.min" [function]
2 rule uint256.min => 0
3
4 syntax Int ::= "uint256.max" [function]
5 rule uint256.max => 2 Int256 - 1

```

Furthermore, rules can be made conditional with `requires`. Combined with the previous definition of the integer value range, we can now describe integer wrap-around.

```

1 syntax Int ::= uint256.chop(Int) [function]
2
3 rule uint256.chop ( I:Int ) => I
4     requires I >=Int uint256.min andBool I <=Int uint256.max
5
6 rule uint256.chop ( I:Int ) => I %Int pow256
7     requires I <Int uint256.min orBool I >Int uint256.max

```

Not that `uint256.chop` is a function that takes a single term of type `Integer` and reduces it to the defined range. The first rule defines that integers are returned as is, so long as they are within the defined range, while the second rule defines the behaviour of overflow as well as underflow. The `uint256.chop` is similar to the `chop` function described in [Hildenbrandt et al., 2018], where they state that this definition serves the purpose of computational optimisation using the \mathbb{K} -built-in functions like `%Int`, whilst skipping the modulo operation for values within the `uint256.chop` value range.

With the EVM being a stack-based machine, stacks are an important part of the KEVM data structure definition. `:` is defined as a typical cons operator, while `++` defines stack concatenation.

```

1 syntax Stack ::= ".Stack" | Int ":" Stack | Stack "++" Stack [
   function]
2
3 rule .Stack ++ WS => WS
4 rule (W : WS1) ++ WS2 => W : (WS1 ++ WS2)

```

`.Stack` acts as an empty element as defined per the first rule and concatenation of elements and stacks is defined in the second rule, both combined inductively defining the KEVM stack data structure. [Hildenbrandt et al., 2018] notes that functions in \mathbb{K} are akin to functional programming.

3.1.2 Defining possible EVM states in \mathbb{K}

To represent the state of the EVM, an XML-style markup notation is used. The encapsulating tags tell us what information is contained within the cell, with some nested

structures. KEVM, however, allows for not only the definition of static, constant values but also that of transitions between states. This feature is key for writing tests in which one wants to observe some transitional property, e.g., the balance or storage of an account, but also to steer execution within a set of given constraints to either succeed or run into an expected, provable failure.

As with the actual EVM, KEVM separates its state into two categories: execution state and network state. The execution state holds the properties and information relevant to only the current transaction's execution. On the other hand, the network state represents the global state of the EVM, in particular, account information.

Execution State

The following example highlights some parts of the KEVM state configuration. Only some static properties are displayed, as a full configuration definition of execution and network state has over seventy cells.

`<evm>` and `<callState>` encapsulate our relevant execution state properties. `<program>`, for example, would contain the relevant bytecode of a smart contract for execution, represented as a map of program counters to opcodes. The `<id>` would be for the account that is responsible for performing the actual execution, while `<caller>` would be the one responsible for the start of the transaction, often the same. `<callData>` defines the ABI encoded function signature used to specify a smart contract's entry point for execution, and `<callValue>` is the amount of Ether (in Wei) sent with the transaction.

```

1 <evm>
2   <callState>
3     <program> .Map </program>
4     <id> 0 </id>
5     <caller> 0 </caller>
6     <callData> .ByteArray</callData>
7     <callValue> 0 </callValue>
8     ...
9   </callState>
10  ...
11 </evm>

```

Network State

Within the network cell of the state configuration, the global state of the EVM is represented. Most importantly, this part allows us to define states for accounts, both privately owned accounts and externally owned accounts. Within the `accounts` cell, we can define as many distinct accounts as needed, using `account`. Here we can define the common account properties like its address with `acctID` and the amount of Ether it is holding with `balance`. `code` and `storage` are only relevant for contract accounts, where `code` holds the bytecode and `storage` the contract's internal state, like the owner's address, ERC20 token balances within a map, etc.

```

1 <network>
2   <accounts>
3     <account>
4       <acctID> 0 </acctID>
5       <balance> 0 </balance>
6       <code> .WordStack </code>
7       <storage> .Map </storage>
8       <nonce> 0 </nonce>
9     </account>
10  </accounts>
11  ...
12 </network>

```

3.1.3 Towards writing Tests and Proofs

Using the basic KEVM layout, let us take a closer look at how one can use these definitions to write tests and proofs in KEVM.

General Structure

The general proof layout is as follows:

```

1 claim
2   <kevm>
3     <k> #execute => #halt </k>
4     <exit-code> 1 </exit-code>
5     <mode> NORMAL </mode>
6     <schedule> ISTANBUL </schedule>
7     <useGas> true </useGas>
8     ...
9     <ethereum>
10    <evm>...</evm>
11    <network>...</network>
12  </ethereum>
13 </kevm>
14 requires ...

```

With `claim`, we indicate to KEVM that we want the Z3 solver to provide us with a proof that the provided definitions and constraints hold. In general, a test verifies that after an initial state, the execution of the smart contract will lead to a desired new state. Within the definition of the EVM state of `kevm`, transitional properties are indicated by an arrow (\Rightarrow). In the `k` tag in the example above, a transition is made from the `#execute` state to the `#halt` state. With this, it is indirectly defined that properties in the first state hold at the time of execution, while those after it should hold after the execution has halted successfully. For properties that remain constant, it is sufficient to omit the transitional arrow. For example, `schedule` indicates the EVM version with its associated rules for block validity.

The `ethereum` tag again contains the execution and network state, and after the `requires` keyword, constraints can be made based on the properties within the state. For this, variables are needed.

State and Variables

Using variables, we can instruct the Z3 solver to verify during execution that the tests hold for any arbitrary value of some property. For example, one can now fix the `callState`'s `id` and `caller` accounts to variables `ACCT_ID` and `CALLER`, respectively. These can then be used to pose further constraints upon them. They do not need to be initialised, but it is helpful to constrain their value range to optimise solver runtime. There is also the anonymous variable (`_`). Mostly used as a subterm, it signifies that we do not care about the value.

```

1 <evm>
2   <callState>
3     <id> ACCT_ID </id>
4     <caller> CALLER </caller>
5     ...
6   </callState>
7   ...
8 </evm>
9 <network>
10  <accounts>
11    <account>
12      <acctID> CALLER </acctID>
13      <balance> BAL </balance>
14      <nonce> _ </nonce>
15      ...
16    </account>
17  </accounts>
18  ...
19 </network>

```

Using the `requires` keyword, these constraints can be formalised similarly to the semantic rules of KEVM. For example, the `balance` in the above example can be constrained to a range between 10000 and 30000.

```

1 requires 10000 <=Int    BAL andBool
2           BAL <=Int    30000

```

Combining state transitions, variables and constraints, tests can be defined for validating certain properties before and after execution. In the example in Figure 3.1, the state transition is as before, from execution to successful halt. Now, using `BAL_INIT` as the initial balance and `BAL_AFTER` as the balance after execution, a claim can be made that for some successful execution call, the caller's Ether balance should strictly be smaller after than before (during) the execution.

```

1 claim
2   <kevm>
3     <k> #execute => #halt </k>
4     <exit-code> 1 </exit-code>
5     ...
6     <ethereum>
7       <evm>
8         <callState>
9           <id> ACCT_ID </id>
10          <caller> ACCT_ID </caller>
11          ...
12        </callState>
13        ...
14      </evm>
15      <network>
16        <accounts>
17          <account>
18            <acctID> ACCT_ID </acctID>
19            <balance> BAL_INIT => BAL_AFTER </balance>
20            <nonce> _ </nonce>
21            ...
22          </account>
23        </accounts>
24        ...
25      </network>
26    </ethereum>
27  </kevm>
28  requires BAL_AFTER <Int BAL_INIT

```

Figure 3.1: A simplified KEVM verification test

3.1.4 Proofs in KEVM – a working example

With the examples provided in this section, the workflow of testing a given smart contract in KEVM will be illustrated. First, for the smart contract used in the example, refer to the SimpleToken contract provided in Figure 3.2. It is a simplified ERC20 token contract with some fields and functions excluded for brevity. The full contract is given in Addendum A.1. `_balances` is a mapping, keeping track of user token balances in the contract’s storage. The `balanceOf` function is a getter function for any account to view the amount of tokens a given address is holding. And finally, the `transfer` function enables accounts to send tokens to another address.

Using the command

```

kevm solc-to-k ./solidity/SimpleToken.sol SimpleToken >
  ./bin/simpletoken-bin-runtime.k

```

```

1 contract SimpleToken {
2   mapping(address => uint256) private _balances;
3   [...]
4
5   constructor() {...}
6
7   function balanceOf(address account) external view returns (uint256) {
8     return _balances[account];
9   }
10
11  function transfer(address to, uint256 value) external returns (bool) {
12    _balances[msg.sender] = _balances[msg.sender] - value;
13    _balances[to] = _balances[to] + value;
14    return true;
15  }
16  [...]
17 }

```

Figure 3.2: A simplified ERC20 token

one can compile the Solidity source code to EVM bytecode and generate \mathbb{K} definitions so KEVM can perform symbolic execution and test the contract's bytecode.

```

1 requires "edsl.md"
2
3 module SIMPLETOKEN-VERIFICATION
4   imports public EDSL
5
6   syntax Contract ::= SimpleTokenContract
7   syntax SimpleTokenContract ::= "SimpleToken" [symbol(), klabel(
8     contract_SimpleToken)]
9   rule ( #binRuntime ( SimpleToken ) => #parseByteStack ( "0x..."
10    ) )
11  syntax Field ::= SimpleTokenField
12  syntax SimpleTokenField ::= "_balances" [symbol(), klabel(
13    field_SimpleToken__balances)]
14  [...]
15  rule ( #loc ( SimpleToken . _balances ) => 1 )
16  [...]
17  syntax SimpleTokenMethod ::= "balanceOf" "(" Int ":" "address" ")"
18    " [symbol(), klabel(method_SimpleToken_balanceOf_address)]
19  [...]
20  rule ( SimpleToken . balanceOf ( V0_account : address ) => #
21    abiCallData ( "balanceOf" , #address ( V0_account ) , .
22    TypedArgs ) )
23    ensures #rangeAddress ( V0_account )
24  [...]
25 endmodule
26
27 module SIMPLETOKEN-MAIN
28   imports public SIMPLETOKEN-VERIFICATION
29 endmodule

```

Figure 3.3: Parts of simpletoken-bin-runtime.k

The \mathbb{K} definitions of a solidity contract use the syntax and rule definitions previously discussed. In Figure 3.3, the `SimpleToken` is defined within a new module. Using the EDSL environment of `K`, which contains most of the KEVM definitions and rules, the module extends the token syntax and rules with those of the KEVM for the subsequent consistency check. KEVM's `solc-to-k` also encapsulates the actual contract bytecode `#binRuntime (SimpleToken)`, which uses the EDSL module's `#parseByteStack` function to parse the actual bytecode instructions. `_balances` is defined as a field on lines 9 and 10, and the pointer location to the map's dynamic storage is set to be at index 1 at line 12.

Next, the test definition is required upon which KEVM can reason and attempt to verify validity. The example in Figure 3.4 defines a test for the transfer function. The test aims to prove: *a transfer transfers the specified amount of tokens from the caller to the receiver, such that they receive the full amount, if all constraints are satisfied*. The constraints are threefold: 1. *accounts must strictly differ*, 2. *the amount transferred must be larger than zero* and 3. *the sender cannot send more tokens than they possess*.

Examining this definition, the values for `output` and `statusCode` constrain the solver to a successful execution. `#parseByteStack` in program defines the call state's execution code, i.e. the runtime code of the smart contract. `#computeValidJumpDests` parses the jump locations for the EVM program, needed for symbolic execution. The actual call is represented by the encoding `#abiCallData("transfer" , #address(TO_ID) , #uint256(VALUE))`. `#address(TO_ID)` and `#uint256(VALUE)` are type-defined via this definition, which is necessary for generating the ABI-encoded function call. According to the definition, account `TO_ID` will receive the `VALUE` amount of tokens. Within the `accounts` tag, the smart contract is further defined, notably the storage contents.

`#hashedLocation("Solidity", 1, CALLER_ID) |- VALUE` fixes the dynamic storage for the field at index 1 (i.e. `_balances`) for the map index `CALLER_ID` to `VALUE`. Using the state transition, it is possible to define this value to be transitional for these dynamic storage spaces as well. With this, one can simply define the value of the fields for the caller to be `BAL_FROM = BAL_FROM -Int VALUE`, i.e. they 'pay' the amount and have that much less after the transaction. For the payment receiver at address `TO_ID`, their balance is set to `BAL_TO = BAL_TO +Word VALUE`, receiving the equivalent amount of tokens.

Finally, the constraints. The first seven restrict the value range of the variables, again mainly for performance optimisation. After that, the previously stated constraints are defined: 1. `CALLER_ID ==Int TO_ID`, 2. `VALUE <=Int BAL_FROM` and 3. `VALUE >Int 0`. The last constraint was added for runtime optimisation purposes.

Combining the test spec and the previously compiled Solidity code, the `kompile-spec` command compiles the `K` specifications, as the last step before calling the solver. The Haskell backend handles symbolic execution, so it has to be compiled with the relevant target parameter. The \mathbb{K} -compiled Solidity code is also referenced via the `syntax-` and `main-module` parameters.

```

1  claim
2  <kevm>
3  ...
4  <ethereum>
5    <evm>
6      <output> _ => #buf(32, 1) </output>
7      <statusCode> _ => EVMC_SUCCESS </statusCode>
8      ...
9      <callState>
10     <program> #parseByteStack ("0x...") </program>
11     <jumpDests> #computeValidJumpDests(#parseByteStack ("0x...") ) </
        jumpDests>
12     <id> ACCT_ID </id> //smart contract
13     <caller> CALLER_ID </caller> //call origin
14     <callData> #abiCallData ("transfer", #address (TO_ID), #uint256 (VALUE)
        ) </callData>
15     ...
16   </callState>
17 </evm>
18
19 <network>
20   <activeAccounts> SetItem (ACCT_ID) _:Set </activeAccounts>
21   <accounts>
22     <account>
23       <acctID> ACCT_ID </acctID>
24       <balance> _ </balance>
25       <code> #parseByteStack ("0x...") </code>
26       <storage> #hashedLocation ("Solidity", 1, CALLER_ID) |-> (BAL_FROM =>
        BAL_FROM -Int VALUE)
27 #hashedLocation ("Solidity", 1, TO_ID) |-> (BAL_TO => BAL_TO +Word VALUE)
28 _:Map </storage>
29       <nonce> _ </nonce>
30       ...
31     </account>
32   </accounts>
33   ...
34 </network>
35 </ethereum>
36 </kevm>
37 requires 0 <=Int ACCT_ID      andBool ACCT_ID      <Int (2 ^Int 160)
38 andBool 0 <=Int CALLER_ID    andBool CALLER_ID    <Int (2 ^Int 160)
39 andBool 0 <=Int ORIGIN_ID    andBool ORIGIN_ID    <Int (2 ^Int 160)
40 andBool 0 <=Int TO_ID        andBool TO_ID        <Int (2 ^Int 160)
41 andBool 0 <=Int VALUE        andBool VALUE        <Int (2 ^Int 256)
42 andBool 0 <=Int BAL_FROM    andBool BAL_FROM    <Int (2 ^Int 256)
43 andBool 0 <=Int BAL_TO      andBool BAL_TO      <Int (2 ^Int 256)
44 andBool CALLER_ID /=Int TO_ID
45 andBool VALUE <=Int BAL_FROM
46 andBool VALUE >Int 0
47 andBool #hashedLocation ("Solidity", 1, CALLER_ID) /=Int #hashedLocation ("
        Solidity", 1, TO_ID)
48 endmodule

```

Figure 3.4: A KEVM test

```
kevm compile-spec simpletoken-transfer-success-spec.k --target
haskell --syntax-module SIMPLETOKEN-VERIFICATION --main-
module SIMPLETOKEN-VERIFICATION --output
simpleTransferSuccess/haskell
```

Finally, the call to execute the test can be made using the following command.

```
kevm prove simpletoken-transfer-success-spec.k --definition
simpleTransferSuccess/haskell/
```

If the defined property holds, KEVM returns:

```
PROOF PASSED: SIMPLETOKEN-TRANSFER-SUCCESS-SPEC.transfer.
success
```

To obtain more details during the run, the `kevm` command also has a `verbose` option. To view the proof after the run, save the proof to a specified save directory.

```
kevm prove simpletoken-transfer-success-spec.k --definition
simpleTransferSuccess/haskell/ --verbose --save-directory
kcfgs
```

Then, with the `kcfg` command, the \mathbb{K} Control Flow Graph (KCFG) can be viewed.

```
kevm view-kcfg --verbose kcfgs simpletoken-transfer-success-
spec.k --definition simpleTransferSuccess/haskell/
```

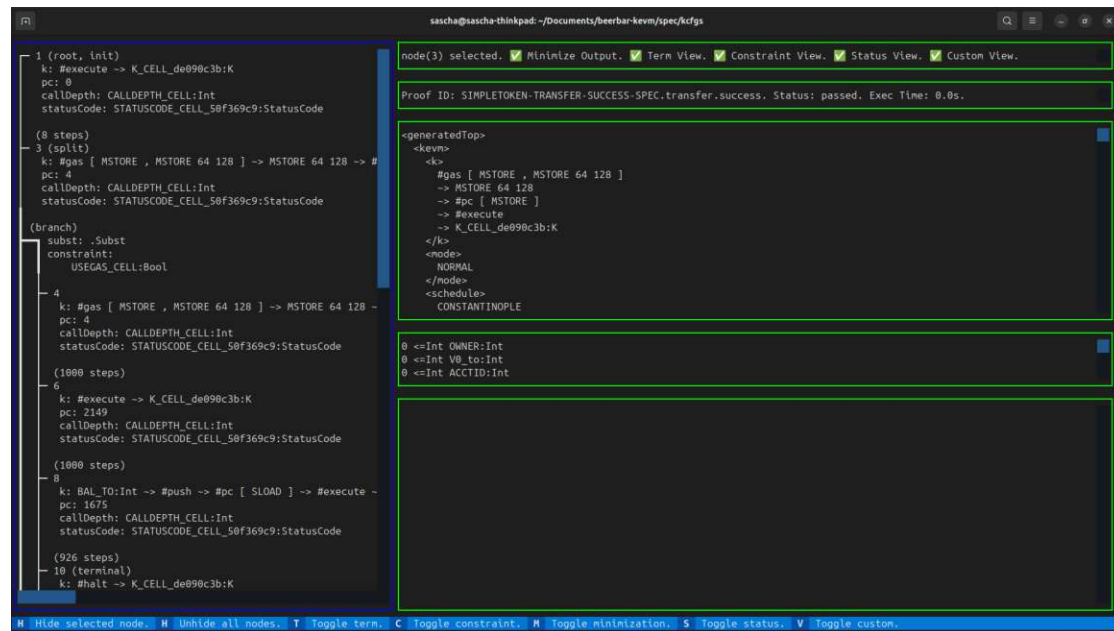


Figure 3.5: The KEVM debug tree

An example is shown in Figure 3.5. The KCFG is a representation of the proof. On the left panel, the execution steps of the proof are shown as a tree. It allows identifying the steps, when branching occurred, etc. The `k` cell shows the current execution relevant to the current node, and `statusCode` the status code, displaying the `statusCode` variable if no final status was reached yet. More detailed information for each node is shown on the right side when selected. There, the current state of the EVM definition is represented as well as the set of all relevant constraints to the symbolic execution path. Should the execution fail at a given step, the `statusCode` will be shown as `EVMC_REVERT` and the failure constraints will be displayed on the right panel. With this, one can investigate failing proofs for faulty constraints to observe faults in either the Solidity code or the test specification.

3.2 ACT

ACT is a specification language that enables the succinct expression of contract behaviour in the form of so-called acts. Each act would then generate one or more KEVM reachability claims. [DappHub, 2019] and [Ethereum Foundation, 2019] lay out the following definitions for the ACT specification language.

3.2.1 Syntax

Since ACT is a higher-level language for KEVM expressions, its semantics are defined by the underlying KEVM backend and the claim generation process, i.e., compilation. Figure 3.6 illustrates the compilation process in more detail.

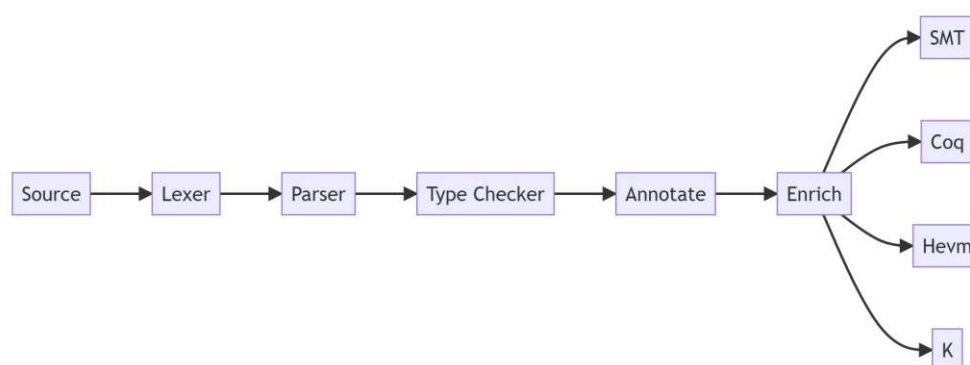


Figure 3.6: ACT Compilation Pipeline

As illustrated, compilation goes through the lexer, parser, type checker, annotation, and enrichment steps. The lexer tokenises the source code for the parser, which constructs an abstract syntax tree (AST) without types. This AST is then type-checked against any typing rules. If successful, a typed AST is generated. The annotation step then

makes any implicit timings explicit, while enrichment adds type-based preconditions to instances, after which they will be sent to the corresponding backend.

Let us take a closer look at the different syntax components.

Types

ACT supports three primitive types: Integer, Boolean, and ByteString. ACT includes the conventional ABI types of the EVM. Integers and integer operations are unbounded and need to be bound for EVM claims. Using ABI types, concise checks can be written; for example, overflow safety for a simple addition.

```
1 iff in range uint
2   x + y
3 returns x + y
```

Sections

Sections allow us to make claims corresponding to a contract state. The two sections described by [Ethereum Foundation, 2019] are `constructor` and `behavior`. In a `constructor` section, claims about the smart contract's creation can be made, while `behavior` is used to make claims about a deployed contract. Sections need to specify an interface that defines the signature of the ABI call to the contract. For `constructor` sections, the interface always refers to the constructor with its corresponding parameters, and `behavior` interfaces allow for any and all defined signatures of the corresponding contract.

Here, a simple example for a `constructor` section:

```
1 constructor of SimpleToken
2 interface constructor(address _owner)
```

It defines the call to the contract constructor of `SimpleToken` with parameter `_owner` of type `address`, after which we can define and make claims about what happens before, during, and after a constructor call. Similarly, the more general `behavior` section can be defined:

```
1 behavior of SimpleToken
2 interface transfer(address to, uint amount)
```

Within the section of this example, a standard ERC20 token transfer call with an `address` where funds will be sent `to`, and the `integer` `amount` can be defined. In order to be able to make further claims within these types of sections, we need further instructions.

Conditionals

These can be added to any section and constrain the behaviour within a section accordingly. Some important conditionals are `iff`, `ensures` and `invariants`.

iff These are knockout requirements, meaning that for both `constructor` and `behavior` sections, the contract creation and calls would end in a `REVERT` if these conditions are not met. `iff` conditionals can be made with simple boolean terms. A line break represents a logical conjunction.

```
1 iff
2   A == B
3   B == C
```

For type ranges like `uint`, `iff` can also be used in the following manner:

```
1 iff in range uint
2   x + y
```

ensures With `ensures` it is possible to specify pre- or post-conditions for `constructor` and `behavior` sections. The example below also showcases the explicit `or` concatenation.

```
1 ensures
2   (post(x) == _x) or (pre(x) == _x)
```

invariants ACT invariants are used to list predicates that hold before and after invocation in `constructor` sections.

```
invariants
  _x = x
  _k = x * y
```

Additional Definitions

To enable more refined proofs and claims, [Ethereum Foundation, 2019] describes further possible definitions. We will list some of these possible definitions within this section.

returns The `returns` definition defines the return value of a `behavior` section's call.

```
1 returns A
```

storage With this definition, we can constrain storage behaviour similar to base KEVM.

```
1 storage
2   balanceOf[CALLER] => balanceOf[CALLER] - value
3   balanceOf[to] => balanceOf[to] + value
```

case Based on the condition, `case` can create claims with different return values. For variables, the return statement needs to define whether the pre- or post-state of the variable is returned.

```
1 case to == CALLER:
2   returns -1
3
4 case to != CALLER:
```

```

5     storage
6         balanceOf[CALLER] => balanceOf[CALLER] - value
7         balanceOf[to] => balanceOf[to] + value
8
9     returns post(balanceOf[CALLER])

```

3.2.2 Proof Example in ACT

For this proof example, we will again use the ERC20 SimpleToken in Figure 3.2. We will illustrate the workflow for an ACT proof according to the original klab documentation [DappHub, 2019]. First, ACT needs to be configured and instructed on where to look for sources and proof specifications.

A standard `config.json` file for configuring the SimpleToken project is structured as follows:

```

1  "src": {
2  "specification": "src/specification.act.md",
3  "smt_prelude": "src/prelude.smt2.md",
4  "rules": [
5    "src/lemmas.k.md"
6  ]
7  },
8  "implementations": {
9  "SimpleToken": {
10   "src": "src/SimpleToken.sol"
11  }
12  },
13 "dapp_root": "dapp"

```

Similarly to generating the proof definitions in KEVM, ACT requires the compiled bytecode. However, in contrast, it was advised to use the `solc` compiler rather than a KEVM internal tool or routine. For our SimpleToken example, the call would look like this:

```

solc --combined-json=abi,bin,bin-runtime,srcmap,srcmap-runtime,
    ast dapp/src/SimpleToken.sol > dapp/out/SimpleToken.sol.json

```

The proof is then defined in the `specification.act.md` as per the config file. With the first two lines, we define the contract and the function call. We then define variables for storage access, `fromBal` and `toBal`, corresponding to caller and receiver balance values which are assigned by the following `storage` section. Furthermore, we are able to add storage state transitions to these assignments. Lastly, the remaining two conditions ensure type safety, i.e. over- and underflow protection and some further constraints to simplify and restrict the solver search space.

```

1 behaviour of SimpleToken

```

```

2 interface transfer(address to, uint amount)
3
4 types
5     fromBal : uint256
6     toBal : uint256
7
8 storage
9     balance[CALLER] |-> fromBal => fromBal - value
10    balance[to] |-> toBal => toBal + value
11
12 iff in range uint256
13     fromBal - value
14     toBal + value
15
16 iff
17     value > 0
18     CALLER_ID != to

```

With our proof setup fully configured, we can now proceed to execute the verification process by first building the necessary K modules:

```
klab build
```

We then run the proof itself, using the following commands, which will generate the proof artefacts and output the results.

```
klab prove --dump SimpleToken_transfer
```

The proof can also be viewed and debugged using the debug command.

```
klab debug <logfile>
```

3.3 Kontrol

With Kontrol [Runtime Verification, 2023a], we are able to write formal specification tests in an extended Solidity syntax. The Foundry smart contract development toolchain [Foundry, 2023] enables this via the import and extension of a `Test` contract, which is then compiled into bytecode and interpreted by Kontrol with an additional set of K rules to the standard KEVM. Core Foundry concepts such as *assertions* and *cheatcodes* have been implemented in this way. The official Kontrol Github repository [Runtime Verification, 2023b] sets out the exact rules and definitions in its *assert.md*, *cheatcodes.md*, etc. markdown files for those interested in a more in-depth look at how Kontrol works under the hood with regards to its K rules.

3.3.1 Syntax

In contrast to ACT, Kontrol’s approach for higher-level expressions was not to create an abbreviated form of the KEVM syntax, but an integration of Foundry’s syntax and principles into the KEVM environment, as previously stated. We will go into more detail, but in general, assertions remain largely the same, while cheatcodes represent a major paradigm shift as they act as a “replacement” for EVM state fields.

General

In general, writing tests in Kontrol is dependent on the `Test` and `KontrolCheats` contracts in Solidity. To initialise a new test, we create a new contract that inherits from both of these contracts, just as with any other Solidity contract inheritance. We again refer to our `SimpleToken` contract in Figure 3.2 for the following examples. Instantiating a new test for the `SimpleToken` would look something like this.

```

1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.13;
3
4 import "../src/SimpleTokenTest.sol";
5 import "forge-std/Test.sol";
6 import "kontrol-cheatcodes/KontrolCheats.sol";
7
8 contract SimpleTokenTest is Test, KontrolCheats { ... }
```

These contracts can make use of typical Solidity syntax. They can define fields, constants, functions, etc., which can then be used throughout any test definitions declared. Let us extend the `SimpleTokenTest` by a `MAX_INT` constant and a field variable that stores our `SimpleToken` instance. Furthermore, we will define the special `setUp` function, the quasi-constructor for the test, called during test startup. We also define a `hashedLocation` function for easier use and access to hashed locations while writing tests.

```

1 contract SimpleTokenTest is Test, KontrolCheats {
2     uint256 constant MAX_INT = 2**256 - 1;
3     SimpleToken token; // Contract under test
4
5     function setUp() public {
6         token = new SimpleToken(...);
7     }
8
9     function hashedLocation(address _key, bytes32 _index) public pure returns
10        (bytes32) {
11         // Returns the index hash of the storage slot of a map at location `
12         // index` and the key `_key`.
13         // returns `keccak(#buf(32,_key) +Bytes #buf(32, index))
14         return keccak256(abi.encode(_key, _index));
15     }
16     ...
17 }
```

The test definitions are encapsulated using Solidity functions. The function signature directs the solver’s search space, with function parameters as the primary variables. Using the Kontrol command line tool, we could then also directly choose which test case we want to run specifically, instead of the whole `SimpleTokenTest`. Test definitions must be prefixed with “test” to be run as such, for example, we can again define a `testTransfer` with `receiver` and `value` as variables left up to the solver.

```

1 contract SimpleTokenTest is Test, KontrolCheats {
2     ...
3     function testTransfer(address _receiver, uint256 _value) public {
4         kevm.infiniteGas();
5         ...
6     }
7     ...
8 }

```

There are also special instructions for testing, mainly assertions and “cheatcodes”. Assertions enable us to make statements about correctness, and cheatcodes help us steer and constrain the search space. We will elaborate on these two concepts in the following sections. As a preliminary example, the `kevm.infiniteGas()` call is a KEVM-specific cheat that allows us to disregard gas calculations when appropriate.

Cheatcodes

Cheatcodes are a core concept of Foundry [Foundry, 2023]. They allow manipulation of EVM states at different test stages. This is a powerful tool that allows us to write sophisticated tests within the Solidity language. One of the most important cheatcodes is the `prank` function, which allows us to set the caller to any address.

```
1 vm.prank(address(0));
```

We can also define what execution outcome is expected during this test. While Kontrol by default expects successful execution, with the `expectRevert` cheatcode, the expected result can be changed to revert, as well as define the expected execution failure code, for example `stdError.arithmeticError` that occurs on integer under- and overflows.

```
1 vm.expectRevert(stdError.arithmeticError);
```

Assumptions are constraints that must hold during symbolic execution. They help model preconditions, constrain the execution search space, and steer the proof toward critical code points and their imposed constraints. They can also help reduce execution times. In the example provided below, we check two addresses `alice` and `bob` for inequality. Furthermore, the `notBuiltinAddress` function excludes the contracts which hold Foundry and Kontrol functionality to avoid any unintended interactions.

```

1 address alice = address(...)
2 address bob = address(...)
3 vm.assume(alice != bob);
4 vm.assume(notBuiltinAddress(alice));
5 vm.assume(notBuiltinAddress(bob));

```

It is also possible to check events emitted using the `expectEmit` function. By capturing the event using the `ExpectEmit` contract, we can then define what we want to check. With `vm.expectEmit(true, true, false, true)` we check the first and second topic of the event for equality, ignoring a (non-existent) third topic and again checking the provided data field. The check is then performed by emitting the expected value as a new event and then collecting the actual event using `emitter.t()`. If the events do not match, the test will fail.

```

1 event Transfer(address indexed _from, address indexed _to, uint256 _value);
2
3 function testTransferEvent() public {
4     ExpectEmit emitter = new ExpectEmit();
5     vm.expectEmit(true, true, false, true);
6     emit Transfer(address(token), alice, 1234);
7     token.transfer(alice, 1234);
8     emitter.t();
9 }

```

Assertions

Assertions are post-hoc constraints that check whether the current EVM state upholds them. With them, we can model post-conditions that dictate the correctness of our tests. Similar to the `vm.assume` cheatcode we define assertions using the `assertEq` functions. Be aware that this function takes two parameters and performs an equality check, as opposed to the infix notation of the `vm.assume`. In addition, there are other assertion functions like `assertLe` that enable us to check for different forms of inequalities.

```

1 assertEq(token.balanceOf(alice), token.balanceOf(bob));

```

3.3.2 Proof Example in Kontrol

With the base concepts established, we can now write a test for our `SimpleToken` contract in Figure 3.2. In Figure 3.7, we again create a test file `SimpleTokenTest`, much like in the previous section. For the setup function, we only instantiate the token. One consideration that could be made for the setup is that we could create personal accounts for the test scenario, provide them with funds and then run the tests. We opted for a more general approach by generalising the token sender, receiver and amount of the transfer via the `testTransfer` function. Since `SimpleToken` does not perform `delegatecalls`, loops or other complex call structures, gas calculations are set to be disregarded via the `infiniteGas` function. We then restrict these generalisations via assumptions, by excluding certain unwanted and nonviable addresses, establishing values to be integer-bound safe and some consistency checks, so we do not start from an already inconsistent contract state. We then calculate the correct and expected values of the balances after the fact and start the token transfer by first using the `prank` cheatcode to assign `alice` as the caller and then perform the transfer call. Finally, we can perform consistency checks based on the previously calculated values and the current contract storage state

after execution via the last two assertions. If both assertions hold and the call execution does not revert, the test instance will be successful.

To run this proof, Kontrol needs to build the required \mathbb{K} files. For this, simply run:

```
kontrol build
```

This, as per the documentation [Runtime Verification, 2023a], first calls `forge build`. Then, artefacts created by `solc` are adapted, similar to base KEVM, into KEVM modules with their corresponding productions and rules, for example, the ABI and bytecode. After this, we are already able to start the proof. With the following command, we can call the `testTransfer` test directly.

```
kontrol prove --match-test SimpleTokenTest.testTransfer
```

To inspect our proof, we can observe the results by using the `list` command. To inspect the proof more thoroughly, kontrol also includes KEVM's KCFG with the `view-kcfg` command. An example of how to use both is given below. The KCFG itself does not differ in presentation from KEVM, refer to Figure 3.5 from the previous section for a KCFG example.

```
kontrol list
kontrol view-kcfg 'SimpleTokenTest.testTransfer(address,uint256
)' --version 0
```

```

1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.13;
3
4 import "../src/SimpleTokenTest.sol";
5 import "forge-std/Test.sol";
6 import "kontrol-cheatcodes/KontrolCheats.sol";
7
8 contract SimpleTokenTest is Test, KontrolCheats {
9     uint256 constant MAX_INT = 2**256 - 1;
10    SimpleToken token; // Contract under test
11
12    function setUp() public {
13        token = new SimpleToken();
14    }
15
16    function testTransfer(address alice, address bob, uint256 amount) public {
17        kevm.infiniteGas();
18
19        uint256 totalSupply = token.totalSupply();
20        uint256 balanceAlice = token.balanceOf(alice);
21        uint256 balanceBob = token.balanceOf(bob);
22
23        vm.assume(alice != bob);
24        vm.assume(alice != address(token));
25        vm.assume(bob != address(token));
26        vm.assume(notBuiltinAddress(alice));
27        vm.assume(notBuiltinAddress(bob));
28        vm.assume(amount > 0);
29        vm.assume(amount < MAX_INT);
30        vm.assume(totalSupply > 0);
31        vm.assume(totalSupply < MAX_INT);
32        vm.assume(balanceAlice > 0);
33        vm.assume(balanceAlice < MAX_INT);
34        vm.assume(balanceBob > 0);
35        vm.assume(balanceBob < MAX_INT);
36        vm.assume(balanceAlice > amount);
37        vm.assume(totalSupply > amount);
38        vm.assume((balanceBob + amount) < MAX_INT);
39        vm.assume((balanceAlice + balanceBob) < totalSupply);
40
41        uint256 newBalanceAlice = (balanceAlice - amount);
42        uint256 newBalanceBob = (balanceBob + amount);
43
44        vm.prank(alice);
45        token.transfer(bob, amount);
46
47        assertEq(token.balanceOf(alice), newBalanceAlice);
48        assertEq(token.totalSupply(), totalSupply);
49    }
50 }

```

Figure 3.7: A Kontrol test for SimpleToken

Verification using Symbolic Execution in KEVM

Symbolic execution in KEVM is a powerful tool for formally verifying the correctness of smart contract behaviour. In this section, we use KEVM to formally verify properties in the context of Ethereum.

4.1 CVE Reports

The Common Vulnerabilities and Exposures (CVE) platform is a publicly maintained database that catalogues cybersecurity vulnerabilities. Since the introduction of Ethereum smart contracts, CVE has begun tracking vulnerabilities such as reentrancy, integer overflows and access control flaws. The reports raise awareness of recurring security pitfalls and are a valuable resource for developers. Jisu Park [SooHo.io, 2023] detected several false reports, i.e., reports about bugs that were actually none, and gave informal proofs, showing the correctness of the programs in question.

4.1.1 Problem

The reports concern a set of four smart contracts:

- Easy Trading Token (ETT)¹
- Pandora (PDX)²
- Bittelux (BTX)³

¹<https://nvd.nist.gov/vuln/detail/CVE-2018-13113>

²<https://nvd.nist.gov/vuln/detail/CVE-2018-13144>

³<https://nvd.nist.gov/vuln/detail/CVE-2018-13326>

- ChuCunLingAIGO (CCLAG)⁴

The reports state that each of these smart contracts suffers from a potential integer overflow for calls made to their `transfer` and `transferFrom` functions. The contracts themselves all share a common interface. They are standard Ethereum tokens with some differences in implementation. We present the smart contracts, their critical code features, and the corresponding formal verification challenges. The full implementations are provided in the Addendum (see Addendum A.1). Here, we only include essential fragments, particularly constructors, transfer-related logic, and key differences affecting verification.

StandardToken Base Contract

The `StandardToken` provides shared ERC20 functionality and is used as a base contract for all four of the discussed token smart contracts. It is split into two contracts, the `Token` and the `StandardToken`.

```

1 contract Token {
2   function totalSupply() external virtual returns (uint256 supply) {}
3   function balanceOf(address _owner) public virtual returns (uint256 balance)
4     {}
5   function transfer(address _to, uint256 _value) public virtual returns (bool
6     success) {}
7   function transferFrom(address _from, address _to, uint256 _value) public
8     virtual returns (bool success) {}
9   function approve(address _spender, uint256 _value) public virtual returns
10    (bool success) {}
11  function allowance(address _owner, address _spender) public virtual
12    returns (uint256 remaining) {}
13
14  event Transfer(address indexed _from, address indexed _to, uint256 _value);
15  event Approval(address indexed _owner, address indexed _spender, uint256
16    _value);
17 }

```

The `Token` contract serves as a syntactic base. It defines the ERC20 functions and their signature in addition to common events. Functions are marked as `virtual` to enable the `StandardToken`'s override of these abstract functions and implementation of the token contract logic. Among them, the critical part for verification is the `transfer` function.

```

1 function transfer(address _to, uint256 _value) public override returns (bool
2   success) {
3   if (balances[msg.sender] >= _value && _value > 0) {
4     balances[msg.sender] -= _value;
5     balances[_to] += _value;
6     emit Transfer(msg.sender, _to, _value);
7     return true;
8   }
9 }

```

⁴<https://nvd.nist.gov/vuln/detail/CVE-2018-13327>

```

7     } else {
8         return false;
9     }
10 }

```

Since we compile these contracts with a recent version of Solidity, some code is slightly modified to ensure compatibility. This modification affects our argument about overflow safety, as newer Solidity versions automatically guard against over- and underflow, introducing the possibility of runtime exceptions. To address this, we define an adapted contract, `StandardTokenUnchecked`, which removes these checks through the `unchecked` block. In the Kontrol section, we verify both the checked and unchecked versions of the contract to obtain a complete proof.

```

1 function transfer(address _to, uint256 _value) public override returns (bool
  success) {
2     if (balances[msg.sender] >= _value && _value > 0) {
3         unchecked {
4             balances[msg.sender] -= _value;
5             balances[_to] += _value;
6         }
7         emit Transfer(msg.sender, _to, _value);
8         return true;
9     } else {
10        return false;
11    }
12 }

```

This structure and its constraints are central to the KEVM proofs of transfer safety. The implementation provides conditional protection against integer underflow, as the maximum `_value` is bounded by the sender's balance. Although such assurances have been made for the sender, there are no further guard mechanisms for the receiver, as the token funds are then simply allocated to them. The function, therefore, has no explicit checks for an integer overflow for balances.

Bittelux

Let us first inspect the Bittelux token A.1. Contract initialisation influences integer overflow safety, specifically how `totalSupply` is handled.

```

1 constructor() {
2     balances[msg.sender] = 10**28;
3     totalSupply = 10**28;
4     name = "Bittelux";
5     decimals = 18;
6     symbol = "BTX";
7     unitsOneEthCanBuy = 22500;
8     fundsWallet = payable(address(msg.sender));
9 }

```

The constructor mints a massive supply to the deployer. The contract also implements a `payable receive` function that performs an internal token sale:

```

1 receive() external payable {
2     totalEthInWei += msg.value;
3     uint256 amount = msg.value * unitsOneEthCanBuy;
4     require(balances[fundsWallet] >= amount);
5
6     balances[fundsWallet] -= amount;
7     balances[msg.sender] += amount;
8
9     emit Transfer(fundsWallet, msg.sender, amount);
10    fundsWallet.transfer(msg.value);
11 }

```

This function may adjust the `totalSupply` value, for example, through the minting of new tokens, depending on its implementation. In the purchase process, we examine however, `totalSupply` remains unchanged. Instead, only the `fundsWallet` balance decreases, representing a transfer of tokens from the central wallet account defined by the token creator. As a result, `totalSupply` serves as a constant upper bound for integer operations. For contracts that implement such a purchase function, we also construct an unchecked version.

```

1 receive() external payable {
2     totalEthInWei += msg.value;
3     uint256 amount = msg.value * unitsOneEthCanBuy;
4     require(balances[fundsWallet] >= amount);
5
6     unchecked {
7         balances[fundsWallet] -= amount;
8         balances[msg.sender] += amount;
9     }
10
11    emit Transfer(fundsWallet, msg.sender, amount);
12    fundsWallet.transfer(msg.value);
13 }

```

CCLAG

Unlike Bittelux, the constructor of CCLAGA.1 is parameterised. However, we observe a similar situation as is the case with Bittelux, since `_initialAmount` is bounded by the limits of `uint256`, which carries over to `totalSupply`

```

1 constructor(
2     uint256 _initialAmount,
3     string memory _tokenName,
4     uint8 _decimalUnits,
5     string memory _tokenSymbol
6 ) {
7     balances[msg.sender] = _initialAmount;
8     totalSupply = _initialAmount;
9     name = _tokenName;
10    decimals = _decimalUnits;
11    symbol = _tokenSymbol;
12 }

```

A major difference is the lack of a purchase or mint function. This implies that the funds, which were created at contract deployment and assigned to the address of the account which deployed it, can only be transferred from the deployer and therefore the `totalSupply` remains constant.

```
1 fallback() external {
2     revert();
3 }
```

This code snippet shows that CCLAG also further restricts arbitrary function calls and ETH transfers, which strengthens our previous observation on the finality of the available supply.

ETT

ETT is similar to Bittelux regarding base contract, constructor, and receive function, except for minor differences and metadata that do not affect contract behaviour with respect to transfer.

```
1 constructor() {
2     balances[msg.sender] = 10**26;
3     totalSupply = 10**26;
4     name = "Easy Trading Token";
5     decimals = 18;
6     symbol = "ETT";
7     unitsOneEthCanBuy = 40000;
8     fundsWallet = payable(address(msg.sender));
9 }
10
11 ...
12
13 receive() external payable {
14     totalEthInWei += msg.value;
15     uint256 amount = msg.value * unitsOneEthCanBuy;
16     require(balances[fundsWallet] >= amount);
17
18     balances[fundsWallet] -= amount;
19     balances[msg.sender] += amount;
20
21     emit Transfer(fundsWallet, msg.sender, amount);
22     fundsWallet.transfer(msg.value);
23 }
```

The receive function mirrors that of Bittelux, facilitating an exchange of ETH to a token by transferring them from the `fundsWallet` to the sender. Our observations made on Bittelux regarding `totalSupply`, therefore, remain the same for ETT.

Pandora

Pandora, on the other hand, is similar to CCLAG, implementing a fully parameterised constructor and no further function calls for minting or purchasing coins.

```

1 constructor(uint256 _initialAmount, string memory _tokenName, uint8
  _decimalUnits, string memory _tokenSymbol) {
2   balances[msg.sender] = _initialAmount;
3   totalSupply = _initialAmount;
4   name = _tokenName;
5   decimals = _decimalUnits;
6   symbol = _tokenSymbol;
7 }

```

Pandora includes the `approveAndCall` pattern used in multiple contracts, which should usually be formally verified for unexpected reentrancy or call failure. We will take note of it for the upcoming test, if the general formal argument is not strong enough regarding Pandora.

4.2 Formal Argument

The idea of Jisu Park was to prove that the assumption that an overflow is possible is inconsistent with the formal models of `transfer` and `transferFrom`.

```

deployer ..... constant(address)
MAX_UINT ..... constant (uint256; 2256 - 1)
value, totalSupply ..... variable (uint256)
s, r ..... variables (address)
balances[x] ..... term (x: address, returns: uint256)

```

`MAX_UINT` is the maximal value of type `uint256`, `value` the amount to be transferred, `totalSupply` the current amount of tokens minted by the contract, `s` the sender and `r` the receiver. `balance` refers to the balance array of the given smart contract, i.e. the funds ledger holding the amount of tokens held by any given address.

Park's argument uses `totalSupply` as a bound for the balances, since `totalSupply` remains a constant value that can only be smaller than or equal to `MAX_UINT`. This follows directly from the semantics of Solidity, as `uint256` restricts the range of value assignments at the time of constructor execution. Furthermore, we observe that each constructor assigns the initial amount to a single wallet account, which is equal to the `totalSupply`.

From this, we can derive the following invariants we need to show:

$$\sum_{n \in \psi} \text{balance}[n] == \text{totalSupply}$$

$$0 \leq \text{balances}[n] \leq \text{MAX_UINT} \text{ for all } n$$

$$0 \leq \text{totalSupply} \leq \text{MAX_UINT}$$

Using induction, we show that these invariants will hold after every function call made, which alters balances in our smart contracts. Our proof uses the specifications of the

Bittelux smart contract. From there, it is easy to generalise to the other contracts, as most of the relevant sections are defined in the base contract StandardToken. The first critical section is the constructor. We use the constructor assignments as the basis for our induction base.

Induction Base:

After calling the Bittelux constructor, it holds that:

$$A. \forall n \in \psi \setminus \text{msg.sender} : \text{balances}[n] = 0 \quad (\text{IB1})$$

$$B. \text{balances}[\text{msg.sender}] = \text{initial_amount} \quad (\text{IB2})$$

$$C. \text{totalSupply} = \text{initial_amount} \quad (\text{IB3})$$

Trivially, our invariants hold, since `initial_amount` can only be assigned to values within the `uint` range.

Induction Step:

Our induction steps will be determined by the behaviour of each function call. We will need to examine each function as a case. The transfer behaviour can be defined as follows, based on the balances after execution is finished, defined as `balances''`:

1. $\text{balances}''[\text{msg.sender}] = \text{balances}[\text{msg.sender}] - _value$
2. $\text{balances}''[_to] = \text{balances}[_to] + _value$
3. $\text{balances}''[n] = \text{balances}[n]$ for $n \notin \psi \setminus \{\text{msg.sender}, _to\}$

Otherwise:

4. $\text{balances}'' = \text{balances}$

Now we need to show our invariants:

- a. $\sum_{n \in \psi} \text{balances}''[n] = \text{totalSupply}$
- b. $0 \leq \text{balances}''[n] \leq \text{MAX_UINT}$ for all n
- c. $0 \leq \text{totalSupply} \leq \text{MAX_UINT}$

We need to distinguish our proof as follows:

Case transfer-1:

Execution arbitrarily aborts or execution of code that does not impact balances and therefore $\text{balances}'' = \text{balances}$. It follows that:

- a. $\sum_{n \in \psi} \text{balance}[n] = \text{totalSupply} \Rightarrow \sum_{n \in \psi} \text{balances}''[n] = \text{totalSupply}$
- b. $0 \leq \text{balances}[n] \leq \text{MAX_UINT}$ for all $n \Rightarrow 0 \leq \text{balances}''[n] \leq \text{MAX_UINT}$ for all n
- c. $0 \leq \text{totalSupply} \leq \text{MAX_UINT}$ holds trivially since `totalSupply` is constant.

We have shown that the invariants hold, and therefore have shown the correctness for this case.

Case transfer-2:

If `balances[msg.sender] < _value`, the execution aborts and no funds will be transferred. Therefore, no balances will be changed, i.e. `balances'' = balances`.

We have reduced this proof to **case transfer-1** and its correctness proof.

Case transfer-3:

If `balances[msg.sender] ≥ _value` but `_value = 0`, the execution aborts and no funds will be transferred. Therefore, no balances will be changed, i.e. `balances'' = balances`.

We have reduced this proof to **case transfer-1** and its correctness proof.

Case transfer-4:

If `balances[msg.sender] ≥ _value`, `_value > 0`, $\dot{+}$ denotes addition with modulo `MAX_UINT + 1`, $\dot{-}$ denotes subtraction with modulo `MAX_UINT + 1` and `balances'` denotes the result of assigning `balances[msg.sender] = _value`, using modulo subtraction. Then:

$$\text{balances}[\text{msg.sender}] \geq _value \quad (\text{A1})$$

$$_value \geq 0 \quad (\text{A2})$$

$$\text{balances}'[n] = \text{balances}[n] \text{ for } n \neq \text{msg.sender} \quad (\text{L1})$$

$$\text{balances}'[\text{msg.sender}] = \text{balances}[\text{msg.sender}] - _value \quad (\text{L2})$$

Since `balances[msg.sender] ≥ _value`, underflow is not possible, and these assertions hold.

Furthermore, with `balances''` denoting the result of assigning `balances[_to] + = _value`, using modulo addition. Then:

$$\text{balances}''[n] = \text{balances}'[n] \text{ for } n \neq _to$$

and either one of two cases may arise:

(no overflow)

$$\text{balances}''[_to] = \text{balances}'[_to] + _value$$

$$\text{if } \text{balances}'[_to] + _value \leq \text{MAX_UINT}$$

or

(**overflow**)

$\text{balances}''[_to] = \text{balances}'[_to] + _value - (\text{MAX_UINT}+1)$
 if $\text{balances}'[_to] + _value > \text{MAX_UINT}$

With these expressions, we can now identify and express integer overflow within our token contracts formally. Lastly, we need to distinguish two further sub-cases: sender and receiver are identical or strictly distinct.

(**$_to == \text{msg.sender}$**):

Then $\text{balances}''[_to] = \text{balances}'[_to] + _value$ (**no overflow**) is satisfied because:

$$\begin{aligned}
 & \text{balances}'[_to] + _value \\
 &= \text{balances}'[\text{msg.sender}] + _value \\
 &= (\text{balances}[\text{msg.sender}] - _value) + _value \\
 &= \text{balances}[\text{msg.sender}] \\
 &\leq \text{MAX_UINT}
 \end{aligned} \tag{L2}$$

Together, these result in:

$$\begin{aligned}
 & \text{balances}''[n] \\
 &= \text{balances}'[n] \\
 &= \text{balances}[n] \text{ for } n \neq _to \\
 & \text{balances}''[_to] \\
 &= \text{balances}'[_to] + _value \\
 &= \text{balances}'[\text{msg.sender}] + _value \\
 &= (\text{balances}[\text{msg.sender}] - _value) + _value \\
 &= \text{balances}[\text{msg.sender}] \\
 &= \text{balances}[_to]
 \end{aligned}$$

We have shown that $\text{balances}'' = \text{balances}$ and therefore have reduced this proof to **case transfer-1** and its correctness proof.

(**$_to \neq \text{msg.sender}$**):

Then it also holds that $\text{balances}'[_to] + _value \leq \text{MAX_UINT}$ (**no overflow**), because:

$$\begin{aligned}
 & \text{balances}'[_to] + _value \\
 &= \text{balances}[_to] + _value && \text{Subst. (L1)} \\
 &\leq \text{balances}[_to] + \text{balances}[\text{msg.sender}] && \text{Subst. (A1)} \\
 &\leq \text{balances}[_to] + \text{balances}[\text{msg.sender}] + \\
 &\quad \sum_{n \in \psi \setminus \{_to, \text{msg.sender}\}} \text{balances}[n] && \text{Monotonicity, (IB1)} \\
 &= \sum_{n \in \psi} \text{balances}[n] \\
 &= \text{totalSupply} \\
 &\leq \text{MAX_UINT}
 \end{aligned}$$

Invariant (a) holds. It follows:

$$\begin{aligned}
 & \text{balances}''[n] \\
 &= \text{balances}'[n] \\
 &= \text{balances}[n] \text{ for } n \neq _to, n \neq \text{msg.sender}
 \end{aligned}$$

```

balances''[_to]
= balances'[_to] + _value
= balances[_to] + _value

```

Furthermore, we have $\text{balances}'[_to] + _value \leq \text{MAX_UINT}$, so:

```
balances''[_to] ≤ MAX_UINT
```

Due to $\text{balances}[_to], _value \geq 0$, it also holds that $\text{balances}''[_to] \geq 0$.

```

balances''[msg.sender]
= balances'[msg.sender]
= balances[msg.sender] - _value

```

Furthermore, we have

```
balances''[msg.sender] ≥ 0
```

Subst., Monotonicity, (A1, A2)

and

```
balances''[msg.sender] ≤ MAX_UINT
```

Subst., Monotonicity, (A2, L2)

Invariant (b) holds, and (c) holds trivially. We have shown our invariants hold at each step and every outcome of `transfer`. The induction steps for `transferFrom` and `receive` are done analogously. Via induction, we have shown that no overflow is possible. We will continue with generating proofs using KEVM, ACT and Kontrol, where we will create some intermediary proofs for KEVM and ACT and attempt to define a full proof definition with Kontrol.

4.3 KEVM

The following sections describe KEVM test claims for successful `transfer` calls based on the formal argument provided above. Each test encodes constraints for contract behaviour, especially focusing on value transfers and account storage transitions. Full listings are available in the Addendum (see Addendum A.2).

4.3.1 Bittelux

The Bittelux KEVM test checks the correct updating of storage when executing a transfer. The total supply is hardcoded, and ownership/storage assumptions are tightly constrained.

The KEVM test definition above is a formalisation in KEVM syntax of the formal argument in section 4.2. We first fix some general KEVM-related syntax for our verification.

```

1 module BITTELUX-TRANSFER-SUCCESS-SPEC
2   imports BITTELUX-VERIFICATION
3
4   claim [transfer.success]:

```


It serves as the lower bound for our inequality. We then continue to extend the inequality as per Park’s argument and our formalisation with:

```
1 andBool (BAL_TO +Int V1__value) <=Int (BAL_TO +Int BAL_FROM)
```

It restricts the transferred value to the balance of the sender, as well as propagating the lower bound. Next, we restrict the sum of balances to be bounded by the total supply, since Bittelux, as we observed, has a fixed amount of tokens issued, which are then circulated later on via purchasing and transferring.

```
1 andBool (BAL_TO +Int BAL_FROM) <=Int TOTAL_SUPPLY
2 andBool TOTAL_SUPPLY <Int (2 ^Int 256)
```

Lastly, the total supply should be bound within the integer value space, since its constant value is less than the maximum integer value. Should this constraint be broken, it would call into question the integrity of the whole contract.

4.3.2 CCLAG

Similar to Bittelux, CCLAG checks the correct transfer behaviour using KEVM constraints, but it omits hardcoded supply constraints. The value of the variable for the total supply remains linked to the contracts storage, to ensure correctness.

```
1 module CCLAG-TRANSFER-SUCCESS-SPEC
2   imports CCLAG-VERIFICATION
3
4   claim [transfer.success]:
5     <program> #binRuntime (ChuCunLingAIGO) </program>
6     <callData> ChuCunLingAIGO.transfer (V0__to, V1__value) </callData>
7     ...
8   requires
9     ...
10    andBool TOTAL_SUPPLY_LOCATION ==Int #loc (ChuCunLingAIGO.
11      totalSupply)
12    andBool TOTAL_SUPPLY ==Int #lookup (ACCT_STORAGE,
13      TOTAL_SUPPLY_LOCATION)
14    ...
```

Other than that, the only remaining difference is the reference to the binary runtime and module names.

4.3.3 ETT

The ETT specification provides similar conditions but uses a hardcoded total supply again, showcasing how total supply is tightly coupled to transfer conditions for these implementations.

```
1 module ETT-TRANSFER-SUCCESS-SPEC
2   imports ETT-VERIFICATION
```



```

1 contract BitteluxTest is Test, KontrolCheats {
2   uint256 constant MAX_UINT = 2**256 - 1;
3   Bittelux token; // Contract under test
4   event Transfer(address indexed _from, address indexed _to, uint256 _value);
5
6   function setUp() public {
7     token = new Bittelux();
8   }

```

In addition to the Bittelux base contract, we introduce an adapted variant named BitteluxUnchecked. This version duplicates the original code but surrounds arithmetically critical sections, such as the addition and subtraction of token amounts in `transfer`, `transferFrom`, and `receive`, with an `unchecked` block. By doing so, we simulate the behaviour of both older contracts compiled without overflow protection and newer Solidity contracts compiled with built-in arithmetic checks. In this context, we prove that arithmetic operations do not overflow even without SafeMath guardrails, and that SafeMath-protected versions of Bittelux do not revert due to overflow exceptions. Apart from this modification, the contracts, tests, and proof setup remain identical; however, test results must be interpreted with this added context in mind.

With the proof setup completed, we define the test case. The solver verifies correctness for arbitrary sender and receiver addresses (`alice` and `bob`) as well as a transfer amount. We omit gas-related considerations, since the focus lies on the semantics of the `transfer` function, which is structurally simple and does not involve loops or recursion. To constrain the input domain, we exclude invalid or special case addresses, such as the zero address and built-in Foundry/Kontrol addresses, with `notBuiltinAddress`. This restriction ensures that the verification process targets only meaningful execution paths relevant to token transfer behaviour.

```

1 function testTransfer(address alice, address bob, address ceasar, uint256
   amount) public {
2   kevm.infiniteGas();
3
4   vm.assume(alice != address(0));
5   vm.assume(bob != address(0));
6   vm.assume(ceasar != address(0));
7
8   vm.assume(notBuiltinAddress(alice));
9   vm.assume(notBuiltinAddress(bob));
10  vm.assume(notBuiltinAddress(ceasar));
11
12  vm.assume(alice != bob);
13  vm.assume(alice != ceasar);
14  vm.assume(bob != ceasar);

```

The identifier `ceasar` serves a specific role in the formal inductive argument. Within the scope of a single test, it denotes an arbitrary address distinct from both `alice` and `bob`. However, in the broader context of the proof, Kontrol and the solver must demonstrate that every such `ceasar` satisfies the constraints imposed. These constraints function as

universally quantified, thereby linking the behaviour observed in a single test instance to the general requirements of the proof.

For readability and ease of use, we create local instances that store the initial values of the relevant contract fields. These stored values also allow us to verify the correctness of balance updates after execution.

```
1 uint256 oldBalanceAlice = token.balanceOf(alice);
2 uint256 oldBalanceBob = token.balanceOf(bob);
3 uint256 oldBalanceCesar = token.balanceOf(ceasar);
4 uint256 oldTotalSupply = token.totalSupply();
```

Next, we encode assumptions to ensure that the contract state satisfies the formal preconditions before invoking the `transfer` call. Using the local instances improves clarity and helps us define the invariants in a way that avoids false-positive overflows caused by the way assumptions are evaluated. For example, the expression `amount + oldBalanceBob` could lead to such a case if `amount = MAX_UINT` and `oldBalanceBob > 0`, or vice versa. This behaviour will be examined in more detail in the following chapter.

```
1 vm.assume(amount <= oldBalanceAlice);
2 vm.assume(oldBalanceAlice <= oldTotalSupply);
3 vm.assume(oldBalanceBob <= oldTotalSupply - oldBalanceAlice);
```

It should be noted that `oldTotalSupply - oldBalanceAlice` could, in principle, produce an underflow. However, given the setup of the Bittelux constructor and the invariant that the sum of balances is constant and equal to the total supply, such an underflow is not possible if the invariants hold.

We then designate `alice` as the sender of the subsequent `transfer` call, thereby ensuring that the transaction context is correctly established.

```
1 vm.prank(alice);
2 token.transfer(bob, amount);
```

Following the execution of `transfer`, we again store the relevant fields.

```
1 uint256 newBalanceAlice = token.balanceOf(alice);
2 uint256 newBalanceBob = token.balanceOf(bob);
3 uint256 newBalanceCesar = token.balanceOf(ceasar);
4 uint256 newTotalSupply = token.totalSupply();
```

We then assert postconditions consistent with both our KEVM specifications and the formal argument. Since Kontrol implicitly asserts successful transaction completion, our focus lies on validating the resulting state changes. Specifically, we verify that `alice` and `bob` exchange balances correctly while all other balances remain unchanged. To capture this, we use `ceasar` to represent arbitrary addresses distinct from `alice` and `bob`, and we assert that each such `ceasar` retains its original balance. As noted earlier, if the solver produces a valid proof for this test, it follows that no arbitrary `ceasar` (i.e., any address other than `alice` or `bob`) can experience a balance change. This establishes one part of the formal argument, namely that the total sum of balances remains constant.

```

1 assertEquals(newBalanceAlice, oldBalanceAlice - amount);
2 assertEquals(newBalanceBob, oldBalanceBob + amount);
3 assertEquals(newBalanceCeasar, oldBalanceCeasar);

```

At this stage, we are no longer in danger of producing a false-positive overflow through addition, as such a case would imply that `transfer` itself does not work as intended. An overflow at this point would necessarily occur within the execution of `transfer`.

For the second part of the argument, we ensure that the sum of the two updated balances remains constant, even though the individual balances change. The first assertion below captures this by comparing the new balance sum with the old one. If these assertions hold, then the updated balances maintain a constant sum and, since all other balances remain unchanged, any execution of `transfer` leaves the overall balance distribution intact. We also verify that `totalSupply` remains constant.

```

1 assertEquals(newBalanceAlice + newBalanceBob, oldBalanceAlice + oldBalanceBob);
2 assertEquals(oldTotalSupply, newTotalSupply);

```

Finally, we assert the invariants after the `transfer` to ensure that they continue to hold. The conditions are adapted to reflect the contract's updated state following the execution.

```

1 assertLe(newBalanceAlice, newTotalSupply);
2 assertLe(newBalanceBob, newTotalSupply - newBalanceAlice);

```

Taken together, these constraints and assertions yield a proof that guarantees `transfer` does not result in an overflow. However, this proof applies only to the case where `alice` and `bob` are distinct and the amount does not exceed `alice`'s balance. To address the case where they coincide, we introduce a separate test, `testEqualTransfer`. In that proof, we simplify the setup by omitting `bob` and adapting the assertions to `alice`, while leaving `ceasar` unchanged.

```

1 function testEqualTransfer(address alice, address ceasar, uint256 amount)
   public { ... }

```

The preconditions and postconditions can be simplified so that the actual transfer amount no longer needs to be considered in the test. Instead, it is sufficient to check the bound on `alice`'s balance.

```

1 vm.assume(oldBalanceAlice <= oldTotalSupply);

```

To perform a transfer in which `alice` sends tokens to themselves, we invoke the corresponding cheatcode followed by the `transfer` call.

```

1 vm.prank(alice);
2 token.transfer(alice, amount);

```

We then assert that all relevant values remain constant. Under these conditions, the invariants trivially hold.

```

1 assertEquals(newBalanceAlice, oldBalanceAlice);
2 assertEquals(newBalanceCeasar, oldBalanceCeasar);
3 assertEquals(newTotalSupply, oldTotalSupply);
4
5 assertLe(newBalanceAlice, newTotalSupply);

```

Additional tests are constructed using the same setup, but we force execution to halt by introducing the assumption `vm.assume(amount > oldBalanceAlice)`. The definitions for `testTransferFail` and `testEqualTransferFail` are provided in the addendum 6.

For the unchecked tests, we extend the `testTransfer` (i.e., the case where sender and receiver are distinct and execution succeeds) by adding the following assertion. This ensures that no overflow occurs in the updated balances.

```

1 assertLe(newBalanceBob, oldBalanceBob);

```

With this, we complete the proof steps necessary for `transfer`. We next consider the `transferFrom` function. The structure of the tests remains largely the same. In the case where the sender and recipient are identical, we simply adjust the function call, as no further modifications are required. Additional considerations for cases where the caller and the funds holder coincide are unnecessary, since our constraints remain unaffected; such cases behave equivalently to a standard `transfer`.

For cases where all addresses are distinct, except for the caller and the funds holder, we define the test case `testTransferFrom`.

```

1 function testTransferFrom(address alice, address bob, address ceasar, address
   david, uint256 amount) public { ... }

```

In this setup, `david` and `alice` may share the same address, whereas `bob` and `ceasar` must remain strictly distinct from both.

```

1 vm.assume(alice != bob);
2 vm.assume(alice != ceasar);
3 vm.assume(bob != ceasar);
4 vm.assume(bob != david);
5 vm.assume(ceasar != david);

```

From there, the test constraints require only minor adaptations to match the structure of the `transferFrom` call, with `david` acting as the caller, `alice` as the funds holder and `bob` as the receiver.

```

1 vm.prank(david);
2 token.transferFrom(alice, bob, amount);

```

The preconditions and postconditions also change slightly. The most notable adjustment is the additional consideration for cases where the caller and the funds holder are identical. In such cases, we only verify that the caller's remaining balance is constant when they do not coincide.

```

1 if (alice != david) {
2     assertEq(token.balanceOf(david), oldBalanceDavid);
3 }

```

Finally, `receive` is the last remaining function which manipulates balances and must therefore be verified. We omit analogous tests for CCLAG and Pandora, as these contracts do not provide a mechanism for purchasing tokens with ETH. The corresponding test follows the equivalent structure as the `transfer` tests, with the only differences being the use of a generic function call and the substitution of the token amount with an ETH amount.

```

1 vm.prank(alice);
2 (bool sent, bytes memory data) = payable(token).call{value: amount}("");

```

In addition, the transferred token amount is calculated based on the `unitsOneEthCanBuy` contract constant. The resulting constraint and assertion ensure the conservation of the sum of balances holds even under ETH-to-token conversions.

```

1 uint256 tokens_bought = amount * token.unitsOneEthCanBuy();

```

A minor consideration concerns the address roles: `alice` now acts as the receiver, while the sender is always defined by the contract as `token.fundsWallet()`. Consequently, the corresponding checks must be adapted to this setting and incorporated into the preconditions.

```

1 vm.assume((tokens_bought + token.balanceOf(alice) <= token.balanceOf(token.
    fundsWallet()) + token.balanceOf(alice)));
2 vm.assume(token.balanceOf(token.fundsWallet()) + token.balanceOf(alice) <=
    token.totalSupply());

```

We now start generating the proofs using Kontrol and the defined tests. As in the previous cases, we build the sources from test specifications with `kontrol build` and execute `kontrol prove`. Each test produces a positive result, allowing us to assert that the proof holds and that integer overflow does not occur. The complete test definitions are provided in the Addendum. For readers interested in further details, the full proofs are available in our repository⁵, together with the source code, generated artefacts, and related data.

4.5.2 CCLAG, ETT & Pandora

For the three remaining token contracts, the test definitions can be largely reused from the Bittelux template. Only minor adaptations are required, such as adjusting the test setups and linking the appropriate contract source files. In cases where a token purchase function is not implemented, we omit the corresponding tests, since they are not applicable. Beyond these adjustments, the verification process follows the equivalent structure as before: we build the sources, invoke the prover, and generate the full proof

⁵https://github.com/splessberger/thesis_kevm_praxis

for each contract. In all cases, the prover returns positive results, confirming that the properties hold consistently across the different implementations.

Evaluation

In this chapter, we evaluate the examined verification tools with respect to their practical usefulness and user experience. Beyond expressiveness, the evaluation focuses on how KEVM tools perform in realistic use cases, including aspects such as ease of specification, debugging support, execution performance, and overall workflow integration. The goal is to assess not only whether the tools are capable of expressing formal properties, but also how accessible and usable they are for developers and researchers in the field.

5.1 User Experience

We will summarise the experience of learning about and working with the various KEVM environments and tools in this section, with the aim of showcasing what an individual with a relevant background in verification can expect from KEVM and related tools, its capabilities, and which surrounding support structures are in place for users and developers.

5.1.1 Resources and Getting Started

Our starting point for our research on KEVM and its related tools was the `jellopaper.org` website. The KEVM introductory paper [Hildenbrandt et al., 2018], which is also referenced there, provides relevant information to acquire a solid foundation for the core concepts surrounding \mathbb{K} and KEVM. Furthermore, the video recording of a talk by Everett Hildenbrandt¹ provides a concise 20-minute overview of the framework. In the same vein, we recommend the recent talk², in which Hildenbrandt provides more detail and examples.

¹https://www.youtube.com/watch?v=tIq_xECoicQ

²<https://www.youtube.com/watch?v=9PLnQStkiUo>

For a more in-depth understanding of K, KEVM and related tools, we primarily rely on the documentation provided by the developers. For the more established \mathbb{K} framework, the \mathbb{K} Tutorial website³ offers detailed guides on writing \mathbb{K} definitions for various languages. The knowledge provided here can help with understanding how KEVM works under the hood and writing KEVM proofs, as well as constraints, which are mostly \mathbb{K} syntax. The custom Google search bar available on jellopaper.org proved helpful for looking up \mathbb{K} syntax components and understanding their usage through relevant examples. For more in-depth questions and support, there is also a Discord server maintained by the KEVM community that is open to joining. This Discord server provides a direct channel to the developer and user community, making it a useful resource for both newcomers and experienced users facing issues.

Additionally, the `kup` package manager is useful, as it streamlines installation by facilitating setup of the \mathbb{K} and KEVM toolchains and simplifies keeping them up to date, which is particularly valuable since these tools are still actively maintained.

Challenges of KEVM

As an emerging tool, KEVM faced common issues encountered with other emerging technologies. Obtaining accurate, up-to-date information was more difficult than with mature technologies, making documentation and community support crucial for overcoming various technical challenges. We will detail them in this section.

Writing Correct Proofs Formalising proofs in KEVM can initially be challenging due to the framework's low-level nature and verbose syntax. Therefore, we recommend formalising proofs outside KEVM beforehand to streamline importing these definitions into KEVM and avoid becoming overwhelmed by syntactic details too early in the process. This preparation also facilitates re-examining faulty or semantically incorrect proofs during debugging, as it provides a higher-level reference for comparison.

³<https://kframework.org/k-distribution/k-tutorial/>

IDE Support Although there is a \mathbb{K} framework extension for VSCode that adds syntax highlighting, code completion, and LSP support, it has not been updated since 2023. Furthermore, it focuses on \mathbb{K} itself and not the KEVM side, so it only helps with some aspects of writing proofs for KEVM. Compilation, execution and visualisation are also mostly restricted to command-line tools. Some aspects of documentation and CLI usage diverge, notably about Solidity code compilation and \mathbb{K} encapsulation. These functions might have been omitted from the base KEVM and relegated to the Kontrol tool.

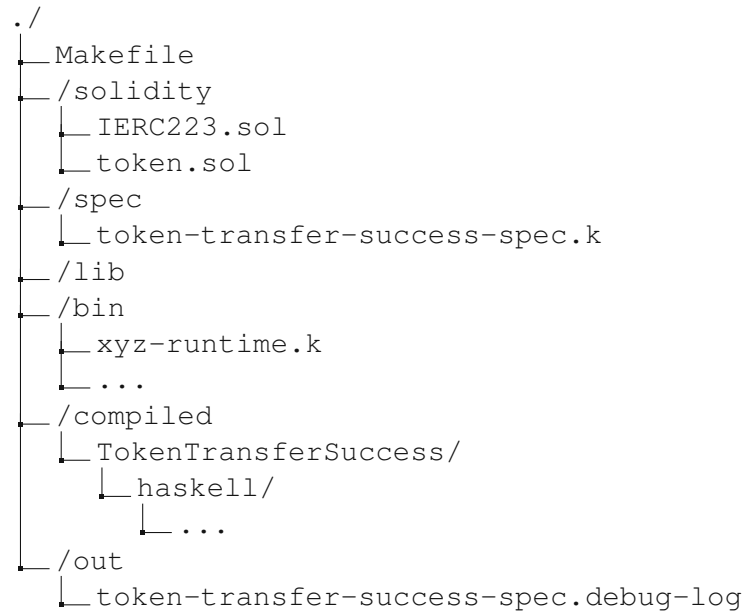


Figure 5.1: Structure of the KEVM test environment folder

Structured approach KEVM can be quite daunting when writing proofs. There is significant syntactic complexity, even in small projects or individual proofs, which can lead to confusion and slow progress. It is therefore recommended to maintain a structured project plan to avoid becoming overwhelmed. Maintaining a consistent structure, naming conventions, and modular decomposition can greatly aid in managing this complexity.

Figure 5.1 presents a proposed project structure, where the `/solidity` folder contains the Solidity source files of the project, i.e., the smart contracts that require formal proofs. The `/spec` directory is intended for KEVM proof definitions, while `/lib` is designated for lemma definitions that can help optimise and reduce prover runtime. The `/bin` folder stores the \mathbb{K} runtime artefacts for each Solidity source, including the compiled bytecode and corresponding \mathbb{K} definitions. Lastly, `/compiled` is the target directory for compiled proof definitions, which are then executed, with the results stored in the `/out` folder.

Proof Scaling Larger proofs as well as small proofs suffer from the same issue: the syntactic overhead required for each proof definition. These large files can confuse users and shift focus away from the important and critical parts of the proof, as well as impact readability. In Figure 5.3, we highlighted the relevant parts for most proofs in green, illustrating the problem. To offset this problem, we can use the overall definition as depicted in Figure 5.3 as a template with tags marked for replacement via curly brackets and populate them from a different spec file, like in Figure 5.2. Sadly, this solution was not well documented and was only discovered by us at a later point in the Verified Smart Contracts repository by the runtime verification team⁴. Furthermore, complex and more involved proofs still require deeper engagement with KEVM for proper modelling and definition.

```
[root]
k: #execute => #halt
gas: #gas({GASCAP}, 0, 0) => _
ensures:

[transfer]
callData: #abiCallData("transfer", #address(TO_ID)
  , #uint256(VALUE))
refund: _ => _
requires:
  andBool 0 <=Int TO_ID
  andBool TO_ID <Int (2 ^Int 160)
  andBool 0 <=Int VALUE
  andBool VALUE <Int (2 ^Int 256)
  andBool 0 <=Int BAL_FROM
  andBool BAL_FROM <Int (2 ^Int 256)
  andBool 0 <=Int BAL_TO
  andBool BAL_TO <Int (2 ^Int 256)
```

Figure 5.2: KEVM compact proof definition

```
// {ROLENAME}
rule
  <k> {K} </k>
  <exit-code> 1 </exit-code>
  <mode> NORMAL </mode>
  <schedule> BYZANTIUM </schedule>

  <ethereum>
    <evm>
      <output> [REDACTED] </output>
      <statusCode> [REDACTED] </statusCode>
      <callStack> _ </callStack>
      <interimStates> _ </interimStates>
      <touchedAccounts> _ -> _ </touchedAccounts>

      <callState>
        <program> #parseByteStack({CODE}) </
          program>
        <jumpDests> #computeValidJumpDests(
          #parseByteStack({CODE})) </
            jumpDests>

        <id> ACCT_ID </id> // contract owner
        <caller> CALLER_ID </caller> // who called
          this contract; in the beginning,
          origin // msg.sender

        <callData> {CALLDATA} </callData>

        <callValue> [REDACTED] </callValue>
        <wordStack> .WordStack -> _ </wordStack>
        <localMem> .Map -> _ </localMem>
        <sp> 0 -> _ </pc>
        <gas> {GAS} </gas>
        <memoryUsed> 0 -> _ </memoryUsed>
        <callGas> _ -> _ </callGas>

        <static> false </static> // NOTE: non-
          static call
        <callDepth> CALL_DEPTH </callDepth>
      </callState>

      <substate>
        <selfDestruct> _ </selfDestruct>
        <log> [REDACTED] </log>
        <refund> {REFUND} </refund>
      </substate>

      <gasPrice> _ </gasPrice>
      <origin> ORIGIN_ID </origin> // who fires tx
      <blockhashes> _ </blockhashes>
    </evm>

    <network>
      <chainID> 0 </chainID>
      <activeAccounts> SetItem(ACCT_ID) _:Set </
        activeAccounts>

      <accounts>
        <account>
          <acctID> ACCT_ID </acctID>
          <balance> _ </balance>
          <code> #parseByteStack({CODE}) </code>
          <storage>
            [REDACTED]
          </storage>
          <origStorage> _ </origStorage>
          <nonce> _ </nonce>
        </account>
        ...
      </accounts>

      <txOrder> _ </txOrder>
      <txPending> _ </txPending>
      <messages> _ </messages>
    </network>
  </ethereum>

  [REDACTED]
```

Figure 5.3: Relevant parts of KEVM proof definition

Many of these issues have been addressed, more or less successfully, with the KEVM-based frameworks ACT and Kontrol.

⁴<https://github.com/runtimeverification/verified-smart-contracts/tree/master>

Debugging We found that debugging in KEVM can be a bit more involved. Simple syntax errors remain largely easy to fix, as the compiler will often catch them. However, debugging failing proofs will involve analysing refutational reasoning. Faulty nodes can be identified using the `kcfg` proof visualisation, as illustrated in Figure 5.4. Here, we can identify the faulty branch via the `EVMC_REVERT` status code, indicating the solver was able to find an execution path that fails. Once the critical sections of a failing proof are identified, one must navigate through the \mathbb{K} -compiled proof definitions, which are often less readable. The biggest challenge will be deciphering and extracting meaning from the refutation. Here, having a formalisation of the proof helps determine whether the fault lies with formalisation, the KEVM definitions or the constraints. Documentation, blog posts and engaging with the Discord forum can help improve understanding at all stages of the process. A solid background in formal verification is also essential.

```

subst: OMITTED SUBST
2 (leaf, target, terminal)
k: #halt -> CONTINUATION:K
pc: PC_CELL 5d410f2a:Int
callDepth: CALLDEPTH_CELL 5d410
statusCode: STATUSCODE_FINAL:St
constraint: { true #Equals ( notB
21
k: JUMPI 3304 bool2Word ( chop
pc: 3290
callDepth: 0
statusCode: STATUSCODE:StatusC
(42 steps)
23 (leaf, terminal)
k: #halt -> CONTINUATION:K
pc: 3303
callDepth: 0
statusCode: EVMC_REVERT

node(21) selected. [x] Minimize Output. [x] Term View. [x] Constraint View. [x] Custom View.
#Not ( { VV1_y_114b9705:Int #Equals 0 } )
{ 0 <=Int CALLER_ID:Int #Equals true }
{ 0 <=Int ORIGIN_ID:Int #Equals true }
{ 0 <=Int NUMBER_CELL:Int #Equals true }
{ 0 <=Int VV0_x_114b9705:Int #Equals true }
{ 0 <=Int VV1_y_114b9705:Int #Equals true }
{ CALLER_ID:Int <Int pow160 #Equals true }
{ ORIGIN_ID:Int <Int pow160 #Equals true }
{ NUMBER_CELL:Int <=Int maxSInt256 #Equals true }
{ VV0_x_114b9705:Int <Int pow256 #Equals true }
{ VV1_y_114b9705:Int <Int pow256 #Equals true }
{ VV0_x_114b9705:Int <=Int ( maxUInt256 /Int VV1_y_114b9705:Int ) #Equals true }
{ false #Equals ( ( notBool VV0_x_114b9705:Int ==Int 0 ) andBool maxUInt256 /Word
VV0_x_114b9705:Int <Int VV1_y_114b9705:Int ) }
( notBool chop ( ( VV1_y_114b9705:Int *Int bool2Word ( ( maxUInt256 /Int
VV1_y_114b9705:Int ) <Int VV0_x_114b9705:Int ) ) ==Int 0 )

```

Figure 5.4: KCFG of a proof unexpectedly resulting in an EVM revert⁵

5.1.2 Challenges and failures of ACT

ACT appears to have been largely abandoned within the KEVM ecosystem in favour of the emerging Kontrol. It still has its use cases for other solvers and frameworks like COQ and HEVM. The initial `k1ab` [DappHub, 2019] project appears to be inactive, as it has not been updated in four years. The new project [Ethereum Foundation, 2019] receives occasional updates, but no new release build has been published since 2021. Nonetheless, we attempted to formalise proofs using ACT. The primary problem ACT was intended to address was syntactic overhead, which it evidently addressed, as proof definitions are considerably smaller than their KEVM counterparts. However, several other issues remain.

Many old issues remain `k1ab` and ACT suffered from the same issues that KEVM had, with the increased complexity of a higher-level language being compiled to the lower-level KEVM proofs. Issues such as a lack of accurate and comprehensive documentation and a complex debugging process remain. This is further exacerbated by the increase in points of failure due to the additional compilation step. Other problems remained unaddressed

⁴<https://docs.runtimeverification.com/kontrol/guides/advancing-proofs/kevm-lemmas>

as a whole. Seeking assistance proved difficult, and inquiries on Discord were met with recommendations to abandon ACT in favour of the newer Kontrol framework.

Discontinued Despite our efforts, we were unable to create a working proof for this thesis. Issues mostly stemmed, as mentioned, from a lack of support structure surrounding the tool, its being largely discontinued and abandoned by the KEVM community, and technological incompatibilities between newer KEVM versions and the older ACT-compiled proof definitions.

5.1.3 Challenges of Kontrol

Kontrol is the most recent and most user-friendly KEVM-based tool. The Foundry approach to writing tests in Solidity significantly simplifies the process of formalising and writing proofs. It also appears to be gaining traction, as the tool itself, related documentation, and the Discord server are active and frequently updated. In particular, the documentation⁶ has been updated extensively and includes concise examples for each step of the programming process. Although Kontrol does a lot of things better than ACT or base KEVM, some challenges remain because it relies on the KEVM base framework.

Structured approach An issue that persists is the overhead associated with the artefacts generated by Kontrol. It is therefore recommended to maintain a well-organised project structure similar to KEVM, as there are multiple technologies to track, including Solidity code, compiled bytecode, and KEVM-compiled proof definitions.

Formal approach While writing proofs in Kontrol is less challenging than in previous tools, strategically defining and formalising proofs remains essential, for the same reasons as in the KEVM base case. In a series of blog posts^{7,8}, Raoul Schaffranek of the runtime verification team highlights such an approach to formalisation from “pen and paper” to a working Kontrol proof.

There, it is noted that the new KEVM-related tool `symbolik`, illustrated in Figure 5.5, is useful for identifying invariants and proof constraints for a given smart contract.

Debugging This point also still suffers from the same issues we faced when working with KEVM, though with increased experience, not as severe. These challenges are also recognised by the development team, and the documentation on how to debug failing proofs¹⁰ has been significantly extended.

False-positives and -negatives To assess the extent to which Kontrol is susceptible to producing “false” proof refutations that arise from inaccuracies or oversights in the test definitions, we examine the pitfalls in our main use case: integer overflow. The following examples illustrate some of these issues.

⁶<https://docs.runtimeverification.com/kontrol>

⁷<https://runtimeverification.com/blog/formally-verifying-loops-part-1>

⁸<https://runtimeverification.com/blog/formally-verifying-loops-part-2>

⁹<https://symbolik.runtimeverification.com/>

¹⁰<https://docs.runtimeverification.com/kontrol/tips/debugging-failing-proofs>

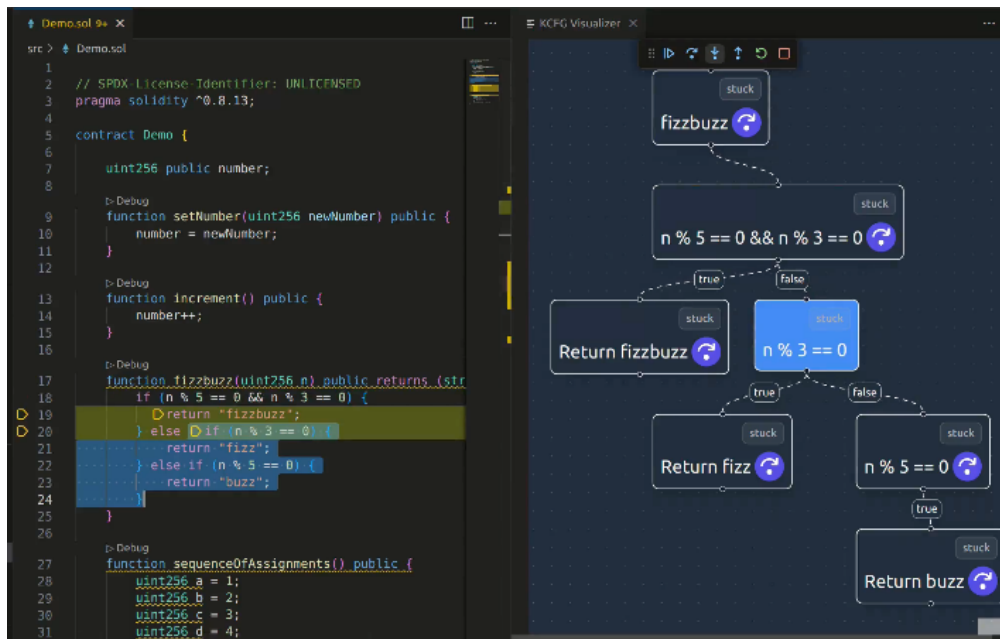


Figure 5.5: Visualization of execution during smart contract debugging using `simbolik`⁹

Our first test highlights the core problem. It fails when `amount = MAX_UINT`, since the addition overflows. This observation underlines an important point: we must argue strictly within the domain and semantics of Solidity, because our assertions are evaluated inside Kontrol/KEVM’s symbolic execution. Kontrol systematically explores all execution paths. If no refutation is found, that is, no revert state is reachable given the contract code and the test assertions, the test passes. When writing test definitions, we therefore need to anticipate where symbolic execution might explore invalid states that would not arise under intended constraints.

```

1 function testOverflow1(uint256 amount) public {
2     uint256 newamount = amount + 1;
3     assertEq(newamount, amount + 1);
4 }

```

This second test investigates how unchecked blocks are treated. Here, the test succeeds, since Solidity’s built-in overflow checks are disabled. However, although the proof holds, the possibility of overflow remains. We therefore need to establish whether such an overflow occurs and, if so, how it affects our reasoning.

```

1 function testOverflowUnchecked1(uint256 amount) public {
2     unchecked {
3         uint256 newamount = amount + 1;
4     }
5 }

```

By adding an explicit inequality assertion, we can detect overflows indirectly. If the addition wraps around, the assertion fails, thereby capturing the overflow condition even though the arithmetic is unchecked.

```

1 function testOverflowUnchecked2(uint256 amount) public {
2     unchecked {
3         uint256 newamount = amount + 1;
4         assertLe(amount, newamount);
5     }
6 }

```

This test case highlights a more subtle issue. An overflow still occurs during the addition, even though the sum is constrained to remain below `MAX_UINT`. The reason is that branch culling occurs only after the addition has been symbolically executed. Thus, the overflow manifests before the solver applies the constraint, and the test fails. This illustrates how the order of symbolic execution and constraint application can create apparent false-positives.

```

1 function testOverflowUnchecked3(uint256 amount1, uint256 amount2) public {
2     vm.assume(amount1 + amount2 <= MAX_UINT);
3     assertEq(amount1, amount1);
4 }

```

To avoid such false positives, we must refine our assumptions. In this case, adapting the assumption ensures that no overflow can occur during the addition, preventing the solver from exploring infeasible branches. This adjustment demonstrates how carefully crafted assumptions are essential to aligning the symbolic execution process with the intended semantics of Solidity.

```

1 function testOverflowUnchecked4(uint256 amount1, uint256 amount2, uint256
2     amount) public {
3     vm.assume(amount1 <= MAX_UINT - amount2);
4     assertEq(amount1, amount1);
5 }

```

5.2 General Tool Results Analysis

We will now present the results of our verification tests. For the base KEVM tests and the ACT, we will provide general remarks on our results. For Kontrol, we will provide further metrics and remarks. Additionally, we will analyse activity within the KEVM community and attempt to review how healthy it is and draw conclusions about the longevity of the tools provided.

All proofs and tests are executed on a standard home or work laptop, a Lenovo ThinkPad E14 with 16GB of DDR4 RAM and an AMD Ryzen 5 with 4,388GHz boost clock speed at six cores with two threads each on an Ubuntu 24.04. operating system. We wanted to see how tests would behave on normal end-user hardware, as that is what is broadly available to consumers and workers in the industry without the added cost of specialised hardware or cloud services.

5.2.1 Baseline Observations of KEVM

The results created with KEVM serve as a semantic reference baseline rather than a performance target. From working with it, we have been able to get a grasp on how KEVM is performing as a backend to both ACT and Kontrol, how expressive the framework can be in its semantic model and how precisely we are able to model the state of arbitrary contracts and the EVM as a whole.

Base KEVM, as shown, possesses a high level of detail in modelling states. It allows detailed specification of even minute aspects of the EVM state, including the EVM world state, personal accounts, and smart contracts (i.e., externally owned accounts). This degree of precision, however, has the drawback of large amounts of boilerplate code, complicating the process of writing and debugging code. Documentation is provided, albeit scattered and with unclear delineations between what knowledge of \mathbb{K} specifically and KEVM in particular was expected of us. Furthermore, since the intended end-user experience is increasingly defined by Kontrol, many functionalities of the base KEVM were adapted to meet that need. The KEVM tool suite was significantly revised, but the documentation was not updated accordingly. We often faced challenges with false usage hints or help texts. Solutions were primarily found in the examples provided in the KEVM source.

Throughout this process, we also familiarised ourselves with the debugging process of KEVM and related tools. KEVM test results are binary; either they pass or are refuted. It requires sufficient user understanding to interpret these results. Did the proof pass because my constraints are not strict enough and do not capture the semantics of the proof correctly? Did the proof fail because the specification is too strict and constrains program flow to perceived semantics that are not originally intended? For these questions, a clean and precise method for converting a specific proof into a KEVM test definition is needed, which again calls upon the user's technical expertise. By learning about the KEVM debugger and KCFG tree inspection, we further deepened our understanding of how the framework operates and what it expects of its users. Investigating these tree structures and faulty proofs helps us not only write better tests but also determine where to expect points of failure, identify limitations of the framework, and navigate unexpected test behaviour in order to arrive at a sound and satisfactory proof.

To conclude, further quantitative assessment was not conducted due to scalability limitations, challenges with changes in tooling, and a shift in focus from using the base KEVM to directly write and generate proofs to an engine that drives more modern tools and languages.

5.2.2 ACT: Tooling Maturity and Ecosystem Viability

As previously noted in earlier sections, support for ACT has been largely discontinued. Nonetheless, we can learn something from attempting to work with the first higher-level language tool for the KEVM framework. Notational overhead has been reduced to a minimum with ACT, making it simpler to approach proofs and definitions in many ways.

While the ACT handbook [Ethereum Foundation, 2019] provides documentation and selected proof examples, its coverage remains limited, as it lacks more comprehensive examples and does not provide complete listings of available functions and parameters. Nevertheless, the reduced syntactic overhead of ACT, combined with prior familiarity with KEVM proofs, allowed for a relatively quick understanding of ACT’s intended usage and verification workflow.

Unfortunately, we were then confronted even faster with the realisation of how outdated ACT actually is. Since ACT must be compiled to KEVM in the tooling pipeline, we encountered a roadblock due to incompatibilities when ACT had to interface with newer versions of the framework. Further research and questions on the official Discord confirmed our experience that it is considered an outdated language for writing proofs in the KEVM framework. We will examine this later on by analysing some public community metrics.

Nonetheless, by investigating ACT, we were able to identify the direction of the framework, with its syntax simplifications and increased documentation. With respect to computational requirements and metrics, we are unable to make qualified assessments given the current state of the tooling suite. An important note regarding ACT and something that affects Kontrol as well is the added development step of building KEVM artefacts with higher-level language imperatives and the target bytecode. This adds another point of failure and computational effort, although the latter is mostly negligible. These compilation layers introduce an abstraction over the KEVM definitions, which obscures debugging, as KCFG primarily displays these KEVM instructions and constraints. In combination with the limited maintenance of the surrounding klab tooling, this layering makes systematic evaluation difficult. Consequently, ACT is best understood not as a failed verification approach, but as a transitional abstraction whose conceptual contributions informed later tools such as Kontrol, while its practical viability diminished as the KEVM ecosystem evolved.

5.2.3 Kontrol Test results

For the evaluation of Kontrol, we again began with general tests based on prototypical ERC20 contracts, gradually extending them to more involved verification scenarios. In contrast to KEVM and similar to ACT, Kontrol enables the definition of such tests with significantly reduced syntactic overhead, allowing the specification effort to focus primarily on the intended semantic properties rather than low-level syntactical details. Due to its integration with the Foundry toolchain and active support within the KEVM ecosystem, more complex test cases can be constructed systematically and incrementally. As a result, Kontrol specifications remain comparatively readable, and the semantic intent of tests and assumptions is more accessible, even as the complexity of verifiable properties increases. With this increased ease of use and the growing support of the KEVM community, Kontrol has become the quasi-standard tool for writing proofs in the KEVM ecosystem, gradually emerging as the preferred approach for defining and maintaining verification specifications. Although some concerns carry over from ACT,

| Checked | Unchecked |
|-----------------------|-----------------------|
| testEqualTransfer | testEqualTransfer |
| testEqualTransferFail | testEqualTransferFail |
| testTransfer | testTransfer |
| testTransferFail | testTransferFail |

Table 5.1: Bittelux Transfer Tests

such as obscured debugging with KCFG, tools like `symbolik` can provide additional support during debugging and are actively addressing usability and adoption challenges.

CVE Test results

We will again analyse the Bittelux ERC20 contract as the prototypical contract for all the relevant CVE tests. Using the previously defined tests, we provide a proof for the `transfer` call consisting of eight individual test functions, four each for the checked and unchecked case. We first want to determine whether we can produce a valid proof that ensures all test cases pass. Furthermore, we want to see how the proof execution performs in terms of runtime.

To reiterate, the proof consists of the tests listed in table 5.1.

These tests are then added to a separate Kontrol build folder and environment. We rebuild the Kontrol artefacts there and are then able to run the full test suite with only our desired tests. The proof execution returns the following result.

```
0:10:39 Multi-proof Mode (4 workers) Finished 8/8 completed. 8
  passed. 0 failed.
```

With 4 worker threads, the full proof finishes in 10 minutes and 39 seconds. All test functions return a positive result. As discussed previously, these 8 tests are sufficient for a full proof regarding integer overflows for the `transfer` function. We further examine the runtime of the individual tests in tables 5.2 and 5.3.

| Test | Result | Time |
|-----------------------|--------|--------|
| testEqualTransfer | PASSED | 3m 42s |
| testEqualTransferFail | PASSED | 3m 49s |
| testTransfer | PASSED | 5m 37s |
| testTransferFail | PASSED | 5m 36s |

Table 5.2: Bittelux Test Results

Table 5.2 presents the case where `MathSafe` integer EVM operations are assumed to be in place. Compared with the case where we explicitly set integer operations to unchecked, we observe a significant runtime speedup on the regular tests. These results are in line

with expected behaviour based on KEVM functionality. By explicitly excluding execution paths for the checks, the solver does not need to explore these branches for erroneous HALT's, leading to the perceived speed up. Indeed, such pruning of branches is a major factor in speeding up proofs by including lemmas. Given the current execution time and hardware, further speeding up the proof execution process was deemed unnecessary, but may warrant consideration as scaling up.

| Test | Result | Time |
|-----------------------|--------|--------|
| testEqualTransfer | PASSED | 3m 50s |
| testEqualTransferFail | PASSED | 3m 49s |
| testTransfer | PASSED | 4m 41s |
| testTransferFail | PASSED | 4m 01s |

Table 5.3: BitteluxUnchecked Test Results

5.3 Community Support

Community support plays a crucial role in the practical usefulness and long-term viability of formal verification tools, particularly in research-driven ecosystems such as KEVM. Beyond the theoretical soundness of a tool, factors such as documentation quality, maintenance activity, and responsiveness within community channels directly influence the feasibility of adoption and sustained use. This section examines community support for KEVM and its associated tooling by analysing publicly available indicators, including repository activity and communication patterns within developer discussion platforms. The resulting observations provide context for the evaluation of KEVM, ACT, and Kontrol, complementing the technical results presented earlier in this chapter.

5.3.1 Github Activity

Public repository activity on GitHub provides useful insights into the maintenance status and development focus of software projects. Metrics such as issue tracking, commit frequency, and pull request activity offer us information on whether a tool is actively developed, in maintenance mode, or effectively abandoned, as discussed by [Coelho et al., 2020]. In this subsection, GitHub activity related to KEVM and its associated tools is analysed to contextualise the evaluation results and to better understand shifts in development effort within the KEVM ecosystem.

The following GitHub repositories are selected for community analysis:

1. **KEVM:**
github.com/runtimeverification/evm-semantic/pulse/monthly
2. **ACT:**
github.com/ethereum/act/pulse/monthly

3. Kontrol:

`github.com/runtimeverification/kontrol/pulse/monthly`

For each repository, issue, commit, and contributor activity is extracted from GitHub and aggregated on a monthly basis. The exported datasets are then imported into Python for pre-processing and visualisation. Apart from basic relabeling and some cutoffs, no manual modification of the data is performed. All visualisations are generated using `matplotlib` and `seaborn`. The following sections analyse the observed trends in detail.

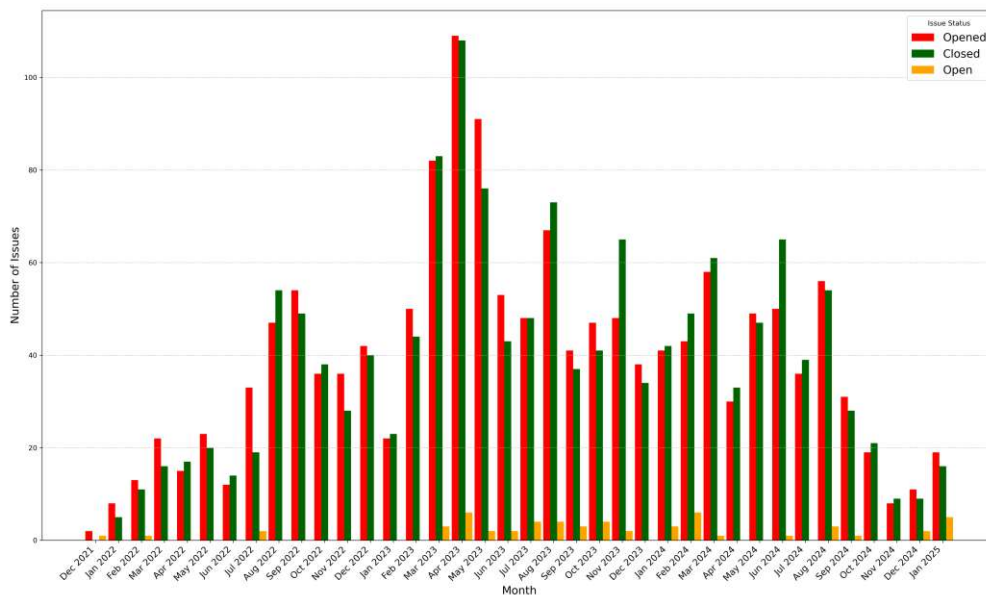


Figure 5.6: KEVM Repository Issue Trends

Issue Trends

Issue activity on GitHub offers insight into both user engagement and the responsiveness of a project’s maintenance process. The number of opened, closed, and unresolved issues can indicate how actively a tool is being used, how effectively problems are addressed, and whether reported issues accumulate over time. In this subsection, issue trends are examined to assess maintenance dynamics and to identify differences in support and sustainability across the tools considered in this thesis.

First, we examine trends in the KEVM repository from December 2021 through January 2025 in Figure 5.6, which show a steady increase in issue activity until mid-2022, indicating growing adoption and user engagement. Throughout this period, the number of closed issues largely follows the number of opened issues, suggesting timely maintenance and active support.

Issue activity peaks in early to mid-2023, with a sharp rise in both opened and closed issues, stabilising thereafter. The close alignment between these values indicates an intensive development and maintenance phase rather than an accumulation of unresolved issues. From late 2024 onward, issue creation declines, while issue resolution remains consistent, and the number of open issues stays low.

Overall, the trend suggests a transition from active development to a more stable maintenance phase, reflecting the toolchain’s maturation rather than a loss of community support.

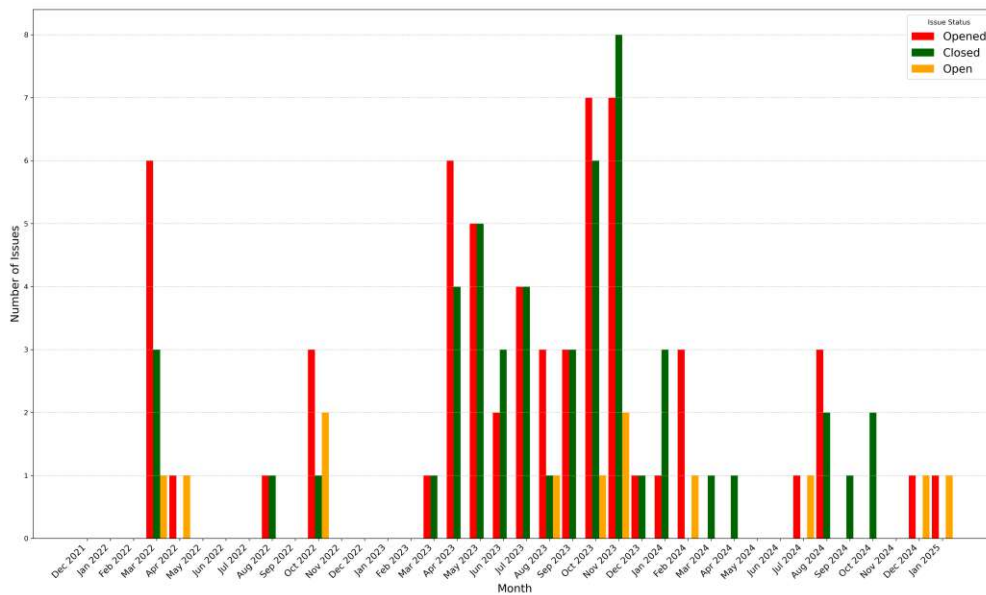


Figure 5.7: ACT Repository Issue Trends

In contrast to KEVM, the ACT graph in Figure 5.7 exhibits substantially lower and more irregular activity throughout the observed period. Issue creation occurs sporadically, with isolated spikes rather than sustained phases of engagement, along with a relatively low number of issues being tracked in the first place, indicating lower interest and push for ACT as an established tool within the KEVM ecosystem. As such, there is no consistent pattern that would indicate ongoing maintenance or a stable contributor base.

The absence of ongoing periods where opened and closed issues closely track each other suggests limited continuous development. Instead, activity appears reactive and intermittent, likely due to individual user reports rather than coordinated maintenance efforts. Furthermore, several periods show no activity at all, reinforcing the impression of reduced adoption and declining community involvement.

Compared to the more sustained and structured issue dynamics observed with KEVM, ACT’s issue trends indicate a tool that is not actively developed or supported. This contrast is consistent with the practical difficulties encountered when using ACT and

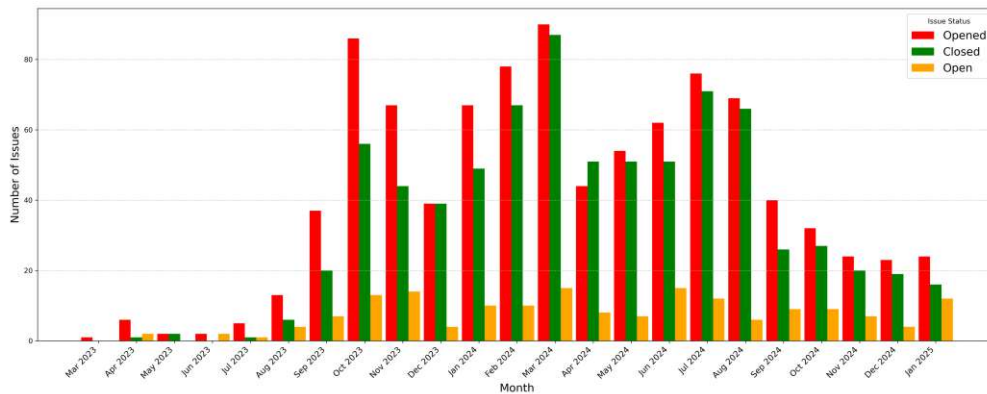


Figure 5.8: Kontrol Repository Issue Trends

supports the interpretation that community attention has largely shifted away from ACT toward more actively maintained tooling.

The issue trends for Kontrol, shown in Figure 5.8, differ heavily from those observed for ACT and align more closely with the sustained activity seen in the KEVM repository. Since Kontrol is a relatively new project, trend data starts in March 2023. Following an initial period of low activity in early 2023, issue creation increases sharply from late 2023 onward, indicating growing adoption and active use of the tool.

Throughout this period, the number of closed issues closely follows the number of opened issues, suggesting a responsive maintenance process. Even during months with more issues being opened, unresolved, open issues remain limited. This indicates that reported problems are addressed in a timely manner by an active development team and individuals contrasting ACT’s sporadic and low-volume issue activity.

The gradual decline in issue volume toward late 2024 and early 2025 indicates stabilisation rather than abandonment, since issue resolution continues steadily. Overall, the issue trends support the interpretation of Kontrol as an actively maintained and community-supported tool within the KEVM ecosystem.

Commit Trends

Commit activity provides further insight into the pace and continuity of software development. Patterns in commit frequency can indicate phases of active feature development, maintenance, or reduced engineering effort. In this subsection, commit trends are examined to complement the analysis of issue activity and to further assess development dynamics and long-term maintenance within the KEVM ecosystem and its associated tools.

The commit activity for KEVM in Figure 5.9 shows stable and continuous development over an extended period, with several peaks corresponding to phases of active feature development and maintenance. Notably, periods of increased commit density align with

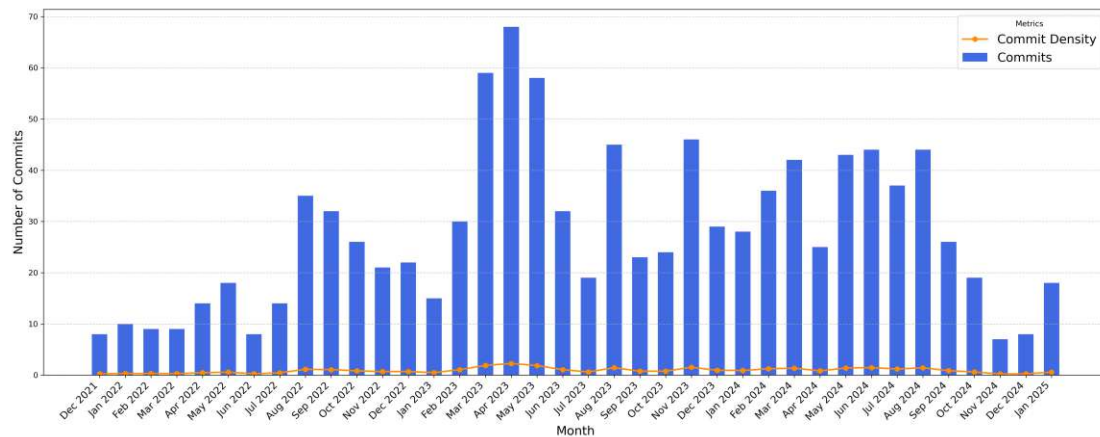


Figure 5.9: KEVM Repository Commit Trends

the heightened issue activity discussed earlier, particularly in early to mid 2023. This correspondence suggests coordinated development and issue resolution efforts rather than isolated contributions. While commit frequency decreases toward late 2024, contributions remain steady, indicating ongoing maintenance rather than stagnation.

In contrast, ACT exhibits low and irregular commit activity throughout the observed period, as shown in Figure 5.10. Commits occur sporadically and are often separated by extended periods of inactivity, with no sustained phases of development. This pattern resembles the previously observed issue trends, reinforcing the impression that ACT is no longer under active development. The absence of consistent commit activity further supports the conclusion that maintenance and feature evolution for ACT have largely ceased.

Kontrol's commit in Figure 5.11 history reflects a comparatively recent and concentrated

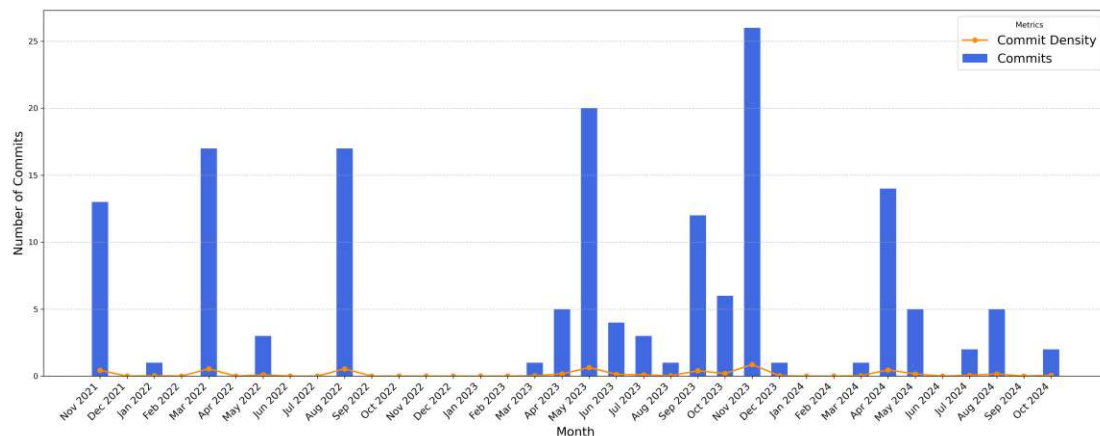


Figure 5.10: ACT Repository Commit Trends

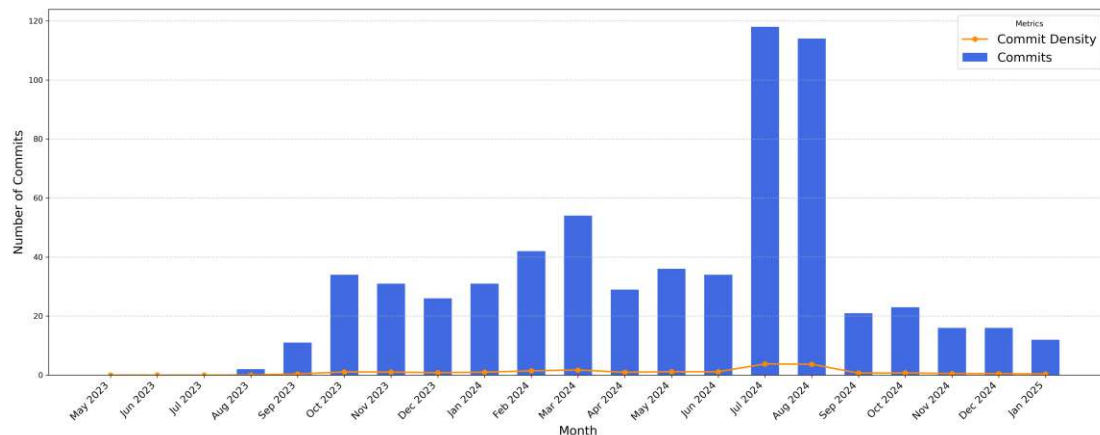


Figure 5.11: Kontrol Repository Commit Trends

development effort. Commit activity increases notably from late 2023 onward, with large peaks in July and August of 2024 indicating active feature development and refinement. These phases coincide with elevated issue activity, suggesting an iterative development process driven by user feedback. Although commit frequency declines slightly toward the end of the observed period, the continued presence of regular commits corresponds to the stabilisation patterns seen in the issue trends, supporting the characterisation of Kontrol as an actively maintained and maturing tool.

Contributor Trends

Contributor trends provide insight into how development effort is distributed within a project. By relating the number of commits per contributor to an approximate commit size, this analysis highlights whether development is concentrated among a small set of core contributors or distributed across a broader community. From this, we can make further inferences regarding the health of the KEVM ecosystem, complementing our previous analysis by determining the degree of centralisation in the development process.

We recognise that the use of publicly available GitHub contributor data raises potential privacy concerns. All analysed information is accessible through public repositories, and contributor names and activity patterns may be used to identify individuals. The presented contributor analysis is intended to describe participation patterns rather than to evaluate or compare individual contributors.

We should note that contributor statistics may be influenced by project-specific development workflows, including pull request merging strategies and automated commits. In particular, maintainers responsible for integrating contributions may appear disproportionately active in terms of commit count or size. As a result, the presented contributor trends should be interpreted as indicative of contribution concentration rather than precise measurements of individual development effort.

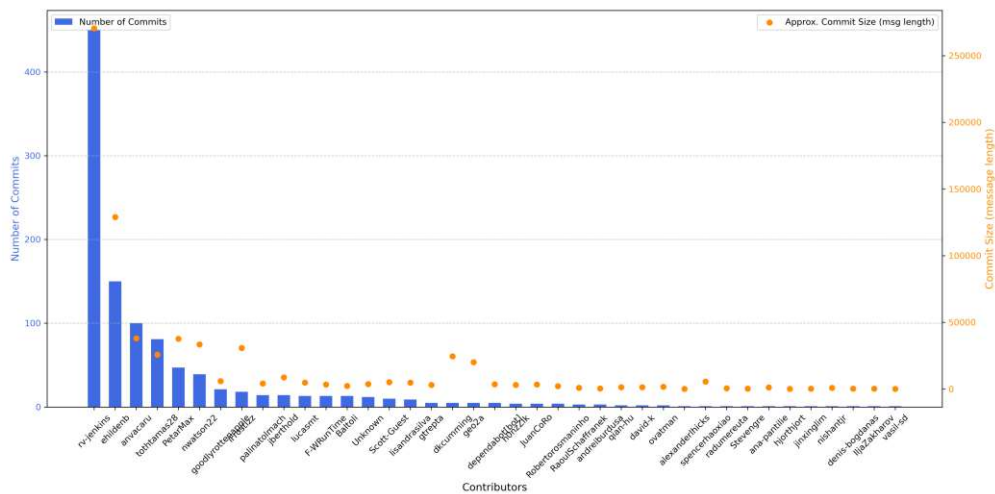


Figure 5.12: KEVM Repository Contributor Statistics

The contributor distribution for KEVM, as illustrated in Figure 5.12, indicates a development process centred around a small group of core contributors, with a long list of occasional contributors. A limited number of individuals account for a substantial share of commits, while most contributors appear only occasionally. This seems to indicate a core team which is responsible for ongoing maintenance and integration of external contributions. As discussed earlier, the high commit counts for top contributors may be partially influenced. However, the overall distribution still indicates a clear core pool of developers.

In contrast to KEVM, ACT exhibits a much smaller contributor base, as shown in Figure

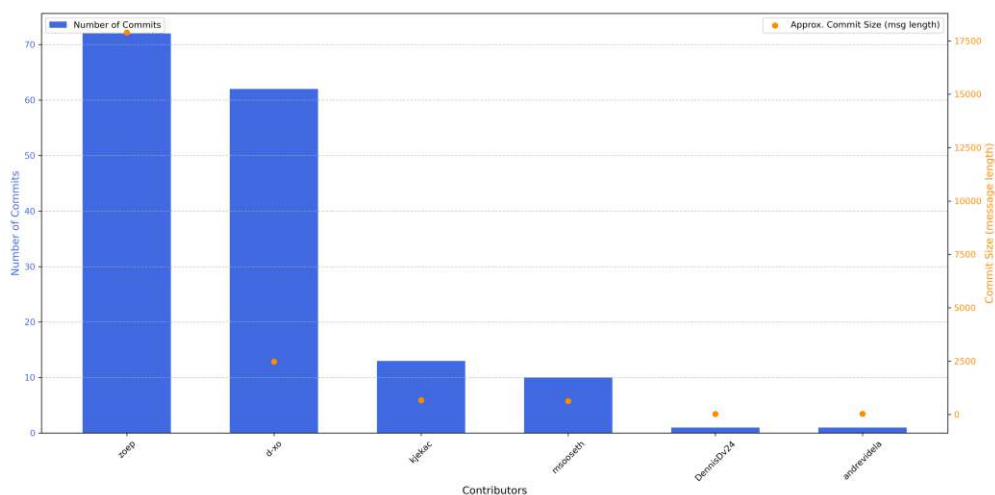


Figure 5.13: ACT Repository Contributor Statistics

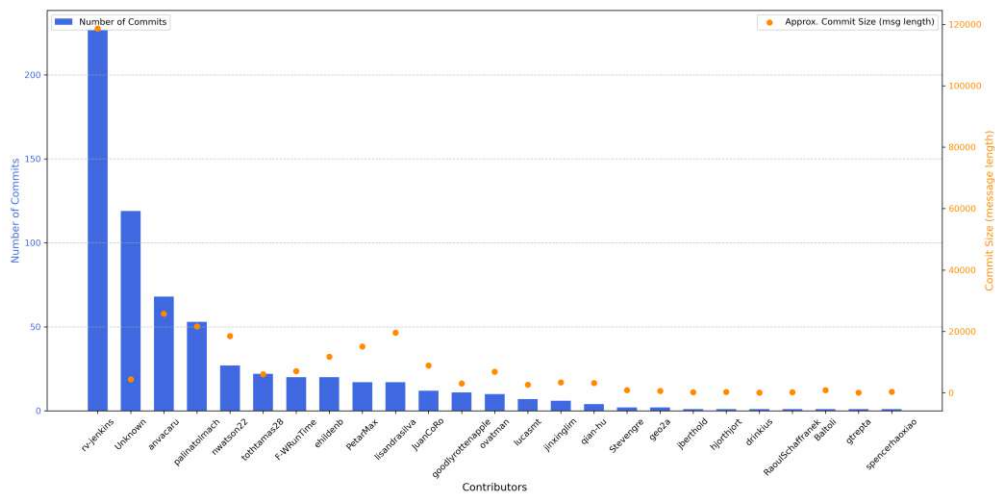


Figure 5.14: Kontrol Repository Contributor Statistics

5.13. Contribution activity consists only of a few individuals, with some contributors appearing only once. The overall volume of commits is low, and there is no clear indication of sustained or continuous contribution over time, consistent with the previously observed commit and issue trends, suggesting that ACT development was limited in scope and duration.

The contributor distribution for Kontrol in Figure 5.14 shows a small group of highly active contributors and a large set of occasional contributors, similar to KEVM. Additionally, multiple contributors appear to be rather active within the observed period. At the same time, development effort remains centred around a limited number of core contributors, consistent with earlier observations. In conclusion, this suggests an actively used and collaboratively maintained codebase.

5.3.2 Discord activity

Discord is used within the KEVM ecosystem as a communication channel for announcements, technical discussions, and user questions, complementing the formal documentation. The empirical study by [Hellman et al., 2022] of open source software forums has shown that messages in these forums and channels can be used to characterise engagement dynamics and support structures within a community. Building on these observations, we will utilise forum participation metrics, keyword-based analyses and related indicators, with the goal of characterising communication patterns and levels of engagement across the KEVM ecosystem.

To this end, chat data is exported from the official *runtimeverification* Discord server¹¹ and stored in a JSON-formatted dataset. The extracted messages are subsequently

¹¹<https://discord.gg/CurfmXNtbN>

analysed using multiple complementary approaches, including user-level activity metrics and content-based analysis. For message content analysis, standard natural language processing and clustering techniques are applied using libraries such as `scikit-learn`, `nltk`, and `gensim`. In addition, message characterisation is performed using pretrained language models. In the following sections, we present the insights derived from these analyses.

Although the analysed Discord channels are publicly accessible, we acknowledge that the use of user-generated content raises potential privacy and ethical concerns. Usernames may function as personal identifiers, and participants may not anticipate their messages being subject to analysis. To mitigate these concerns, the analysis is limited to aggregated statistics and does not include message content or user identifiers beyond those already visible, except official *runtimeverification* staff. The results are intended to characterise patterns within the community rather than individual behaviour.

User Metrics

User behaviour within community communication platforms provides insight into how participants interact with the ecosystem. This section examines user-level metrics to characterise participation patterns and levels of engagement within the community.

The first Figure 5.15 shows the distribution of messages sent per user and shows a skewed pattern. Most users contribute only a small number of messages, while a small minority accounts for a disproportionately large share of total activity. We assume this large spike in users with a small number of messages sent can be explained by developers seeking

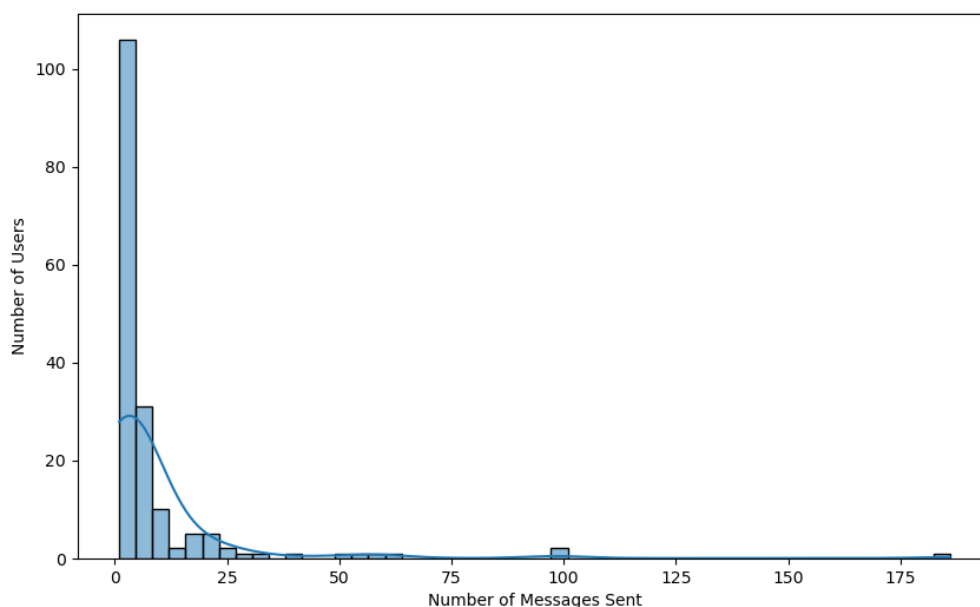


Figure 5.15: Distribution of user-engagement

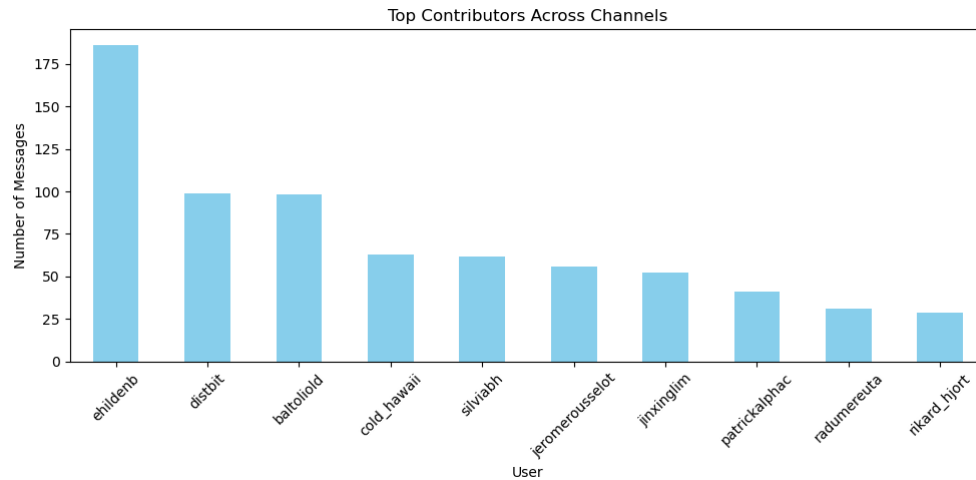


Figure 5.16: Top Contributors

help with a specific problem and afterwards only engaging sporadically with the forum. Furthermore, the diminishing distribution suggests that a percentage of those users and other individuals remain and become more active participants in the community. Lastly, at the tail end, are the top contributors, as shown in Figure 5.16, illustrating the most active contributors across channels, measured by total message count. This subset of participants plays a central role in sustaining discussions, answering questions, and facilitating interaction. From this subset, we identified only Everett Hildenbrandt (*ehildenb*) and Silvia Barredo (*silviabh*) as members of the team. Other accounts seem to be either power users, active members of the community or retired accounts that are no longer identifiable.

Message Content

Beyond user-centred metrics, the content of community messages offers insight into the types of interactions occurring within developer forums. Prior work by [Wu et al., 2024] has shown that forum posts can be systematically characterised and grouped according to their underlying intentions, such as asking questions, providing explanations, or discussing design decisions. We will analyse message content using topic clustering and apply such message characterisation techniques.

Topic Clustering

Topic clustering was performed using standard natural language processing techniques implemented with the *nltk* and *gensim* libraries. The four resulting topics, with a coherence score of 0.669 and a perplexity score of 1132.81, are as follows.

Topic 1: issue test ercx transfer balance contract token call amount address

Topic 2: syntax module error import endmodule exp int rule token test

Topic 3: int rule line info file function true range buf equal

Topic 4: error help using run kevm version get kontrol like file

This indicates that discussion within the analysed channels is primarily centred around practical use and troubleshooting of the tooling. Topic 1 encompasses terms related to smart contract behaviour and ERC token operations. This can suggest discussions focused on contract semantics, test execution, and state-related properties. Topics 2 and 3 contain terminology associated with syntax and rule definitions, typically found in conversations about writing and structuring formal specifications. Topic 4 is characterised by terms related to errors, tool execution, and versioning, indicating discussions involving setup issues and understanding tool behaviour. Together, these topics suggest that message content is largely technical, emphasising specifications, development, debugging, and tool use rather than high-level conceptual discussion.

The coherence score of 0.669 indicates a moderate degree of semantic consistency within topics, suggesting that the extracted clusters capture meaningful themes despite the usually informal nature of Discord. The perplexity score is consistent with the brevity and diversity of chat-based communication, as observed in research on short-text topic modelling for social media and similar platforms [Kinariwala and Deshmukh, 2023]. Overall, the results support the view that Discord discussions function primarily as a technical support and problem-solving forum within the KEVM ecosystem.

Characterization

In addition to topic-based clustering, message content was further analysed through intent-based characterisation following the methodology proposed [Wu et al., 2024]. This approach attempts to classify messages by intent, such as *Learning*, which captures messages aimed at understanding basic concepts or terminology, and *Explicit Error*, which refers to reports of concrete failures or error messages. *How-to* encompasses requests for guidance, whereas *Discrepancy* covers discussions of mismatches between expected and observed behavior. *Conceptual* messages focus on abstract or design level considerations, *Review* on evaluation or confirmation of existing solutions, and *Other* on messages that do not fit into any of the other categories. Such a characterisation was performed using a zero-shot classification pipeline with the pretrained natural language inference model facebook/bart-large-mnli, assigning intent labels without requiring task-specific training data.

The distribution of message categories in Figure 5.17 shows that communication can mainly be categorised into *Other* and *Review*, likely due to the nature of Discord communication culture, with messages consisting of short responses, confirmations, follow-up remarks, or context-dependent exchanges. *Conceptual* and *Discrepancy* messages being also fairly prevalent suggests that discussions frequently go beyond simple error reporting and address higher-level questions about behaviour and semantics of KEVM and its related tools. This corresponds to our identified topic clusters related to specification rules, syntax and contract semantics, indicating that users often discuss their interpretations

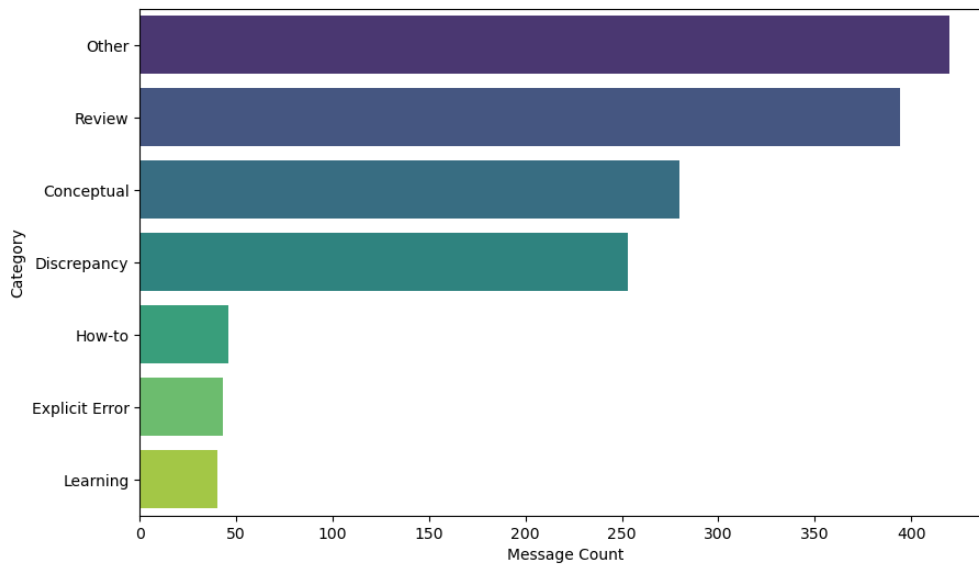


Figure 5.17: Message Characterization as per [Wu et al., 2024]

or diagnoses on these topics. In contrast, *How-to*, *Explicit Error*, and *Learning* messages occur less frequently, suggesting that practical guidance and error reports are present, although they are not typical interactions.

The distribution of message categories over time, illustrated in Figure 5.18, indicates clear phases of community interaction. The first noticeable increase in messages in early to mid 2022, mostly consisting of *Other* and *Conceptual* content. This suggests heightened conversational activity that may be associated with onboarding, coordination, or informal

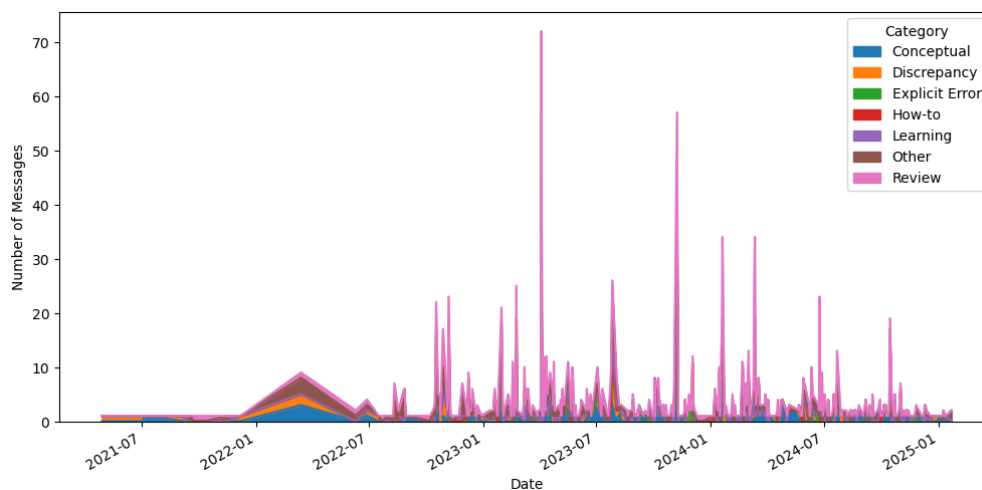


Figure 5.18: Message characterization as per [Wu et al., 2024] over time

clarification, rather than technical discussions. Even larger spikes are visible in the *Review* category during mid 2023 and towards late 2023. These peaks coincide with the periods of increased issue and commit activity observed earlier, again suggesting phases of active development or stabilisation within the KEVM ecosystem, during which changes prompted broader discussion on communication channels. Overall, the category trends indicate that community communication also changes with development phases, transitioning from informal or coordination-focused exchanges toward more review-oriented discussions as the KEVM ecosystem matures and evolves.

Together, the GitHub and Discord analyses provide a complementary view of community activity within the KEVM ecosystem. Repository metrics capture development and maintenance dynamics, while Discord activity reflects informal discussion, support, and coordination. Across both platforms, activity is characterised by contributions concentrated among a relatively small group of highly engaged participants, accompanied by broader, sporadic participation. The combined results suggest that community interaction and support extend development workflows, forming an ecosystem of tooling, discussion, and user assistance.

Conclusion

To conclude, we revisit our research questions, the intent of this thesis, and the results achieved.

This thesis developed a detailed technical background and presents an evaluation based on concrete tool usage, vulnerability-driven test cases, and community activity analysis. This approach provided a practice-oriented perspective, emphasising real-world behaviour, and enabled a detailed examination of the KEVM toolchain at multiple levels with respect to applicability and use cases.

We then evaluated the KEVM ecosystem under different lenses, discussing challenges we personally encountered during our efforts, as well as a qualitative assessment of each of the tools' behaviour and their workflow, followed by some concrete proof results and metrics for within Kontrol for the discussed integer overflow problems stated in the CVE reports that were formally first disputed by [SooHo.io, 2023]. Lastly, we analysed GitHub and Discord forum data sets to gain further insights into the community behind KEVM and its related tools, how active they are with code maintenance as well as technical support and what forms this can take.

To detail our work and findings, let us quickly revisit our research questions.

RQ1. As an individual with specialised education in the field, what difficulties arise when working with the verifier?

Working directly with KEVM exposes users to a high level of semantic detail, which, while enabling precise reasoning, also introduces significant syntactic and conceptual overhead. Constructing and debugging proofs requires familiarity with the underlying operational semantics of KEVM and \mathbb{K} . ACT represents an early attempt to mitigate this issue by introducing a higher-level specification, but limited maintenance and tooling support have rendered it mostly unusable. Kontrol, however, lowers the barrier to entry by embedding verification directly into the Foundry testing workflow, though this increased usability

comes at the cost of obscuring certain low-level semantic controls and complicating debugging through an additional compilation layer.

RQ2. To what extent is it possible to define accurate rules for smart contracts' semantic properties in KEVM, ACT and Kontrol?

All three tools support the formal specification of semantic properties, but at different abstraction levels. KEVM allows for highly precise and expressive definitions, directly via the operational semantics of the EVM. ACT offers a more concise and readable syntax that compiles to KEVM rules, but its abstraction layer introduces additional complexity. Kontrol continues with this approach and shifts the specification process into Solidity, enabling developers to express properties in a familiar language and reuse existing test infrastructure from Foundry. While this improves accessibility, it again obfuscates the level of direct semantic control available in KEVM.

RQ3. How can we prove semantic properties and the absence of certain vulnerabilities, specifically concerning the ERC20 token standard? Specific examples include arithmetical exceptions like overflows and underflows, (in)ability to send ether to and from a contract, etc.

The conducted case studies into general ERC20 token contracts and the disputed CVE reports, as per [SooHo.io, 2023], demonstrate that KEVM tools are capable of formally proving semantic properties and identifying violations for realistic smart contracts. Using ERC20 vulnerabilities as examples, the thesis shows how symbolic execution and semantic reasoning can be applied to formally verify properties such as correct balance updates and safe arithmetic behaviour. The successful verification of the disputed CVE reports further establishes the applicability of these tools to real-world security problems.

RQ4. Given a known semantic property and a smart contract in violation of such a property, what can we deduce other than the violation itself?

Beyond detecting violations, we can trace issues back to their causes using the provided tools, particularly in newer versions. However, they are not yet fully developed. It requires effort and, again, knowledge of the operational semantics of KEVM and \mathbb{K} to adequately interpret proof traces and deduce root causes of failure. However, it highlights that these proof structures are not simply black boxes that return true or false, but can be used to deduce further information by correctly analysing the generated proof traces.

Outlook

Future work could extend the scope of this thesis in several ways. One clear continuation may be to advance the section of evaluating performance of the KEVM tool suite, focusing on creating semantically equivalent proofs for each tool and evaluating them comparatively against other symbolic execution tools similar to KEVM, like *Mythril*, *Manticore* and other formal verification tools like *VeriSol*. To that end, it may also be worthwhile to further investigate scalability concerns and develop larger, more intricate proofs, with a focus on benchmarking more extreme scenarios in terms of execution time and resource cost.

Another possible direction is the systematic analysis of Ethereum vulnerabilities reported in the CVE database. By clustering and classifying these reports according to their semantic failure patterns, it may be possible to identify recurring classes of contract vulnerabilities. Based on such a taxonomy, verification templates or proof obligations could be developed for each of them. This would not only support more scalable verification workflows but would also help bridge the gap between informal vulnerability descriptions and formal semantic properties.

Overview of Generative AI Tools Used

Text and Grammar

Grammar and spelling suggestions were provided by ChatGPT, Grammarly and the integrated AI service of Overleaf, Writefull. ChatGPT responses were then used to adapt existing text passages based on the suggestions within the responses, rather than simply being copied and pasted. The intended use was that of a grammar, spelling and tonal-consistency checker rather than a tool to generate full text passages. With Grammarly and Writefull, that distinction is rather clear, as they were used to suggest changes for existing text passages and directly replace them within the text editor, much like spelling and grammar checks one might be familiar with from Microsoft Office's Word.

Help with Python for Visualisation Tasks

We used ChatGPT to generate boilerplate code and templates for the text analysis section in Chapter 5 to support the creation of accurate visualisations.

Extracting Discord Messages

Prompt Summary

- Load messages from JSON files obtained via the Discord API.
- Extract relevant fields like author, content, timestamp, and replies.
- Handle missing or unexpected data structures.

Final Code

```
1 import os
2 import json
3
```

```

4 def load_messages():
5     data = []
6     for filename in os.listdir('.'):
7         if filename.endswith('_log.json'):
8             with open(filename, 'r', encoding='utf-8') as file:
9                 messages = json.load(file)
10                for message in messages:
11                    data.append({
12                        "Channel_ID": message['channel_id'],
13                        "Author": message['author']['username'],
14                        "Content": message['content'],
15                        "Timestamp": message['timestamp'],
16                        "Replies": len(message.get('reactions', [])),
17                        "Thread": message.get('thread', {}).get('name', 'None')
18                    })
19    return data
20
21 data = load_messages()

```

Listing 6.1: Loading and Processing Discord Messages

Topic Modelling with BERTopic

Prompt Summary

- Identify problem-related messages using keywords.
- Apply BERTopic for clustering.
- Handle empty topic lists and embedding issues.

Final Code

```

1 from bertopic import BERTopic
2 from sklearn.feature_extraction.text import CountVectorizer
3 from sklearn.decomposition import TruncatedSVD
4
5 problem_keywords = ["error", "issue", "problem", "fail", "bug", "crash", "stuck"]
6
7 def filter_problem_messages(messages):
8     return [msg for msg in messages if any(kw in msg.lower() for kw in
9         problem_keywords)]
10
11 def analyze_problematic_channels(data):
12     topic_model = BERTopic()
13     vectorizer = CountVectorizer(stop_words="english")
14
15     for channel in set(d['Channel_ID'] for d in data):
16         messages = [d['Content'] for d in data if d['Channel_ID'] == channel]
17         problem_messages = filter_problem_messages(messages)

```

```

17
18     if len(problem_messages) > 6:
19         X = vectorizer.fit_transform(problem_messages)
20         svd = TruncatedSVD(n_components=min(5, X.shape[0] - 1))
21         X_reduced = svd.fit_transform(X.toarray())
22
23         topics, _ = topic_model.fit_transform(problem_messages,
24                                               embeddings=X_reduced)
25         topic_model.save(f"{channel}_bertopic_model")
26 analyze_problematic_channels(data)

```

Listing 6.2: Filtering and Analyzing Problem Topics

Analysing Replies and Response Times

Prompt Summary

- Identify most active repliers.
- Compute response times for replies.
- Handle cases where no replies exist.

Final Code

```

1 from collections import defaultdict
2 from datetime import datetime
3
4 def analyze_replies(data):
5     response_times = defaultdict(list)
6     user_reply_count = defaultdict(int)
7
8     for message in data:
9         if 'replies' in message:
10            original_time = datetime.fromisoformat(message['Timestamp'])
11            for reply in message['replies']:
12                reply_time = datetime.fromisoformat(reply['timestamp'])
13                response_times[message['Author']].append((reply_time -
14                                                           original_time).total_seconds())
14                user_reply_count[reply['author']]['username'] += 1
15
16     return response_times, user_reply_count
17
18 response_times, user_reply_count = analyze_replies(data)

```

Listing 6.3: Analyzing Replies and Response Times

Visualising Time Tracking Data

Prompt Summary

- Load activity data from CSV.
- Plot hours spent per technology.
- Adjust bar chart to separate activities per technology.

Final Code

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import seaborn as sns
4
5 df = pd.read_csv("timekeeping.csv")
6
7 plt.figure(figsize=(10, 6))
8 sns.barplot(x="Technology", y="Time (hrs)", hue="Activity", data=df, dodge=
    True)
9 plt.xticks(rotation=45)
10 plt.title("Technology Usage Breakdown")
11 plt.savefig("tech_usage.png")
12 plt.show()
```

Listing 6.4: Plotting Activity Data from CSV

Fixing Issues

Stopword Removal Issue

Issue: Default stopwords were not removing common words properly.

Fix: Combine built-in and custom stopwords.

```
1 from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
2
3 custom_stopwords = ["thanks", "thank", "hi", "bye", "hello", "ok"]
4 all_stopwords = set(custom_stopwords).union(ENGLISH_STOP_WORDS)
5
6 def remove_stopwords(text):
7     return " ".join([word for word in text.split() if word.lower() not in
    all_stopwords])
```

Listing 6.5: Fixing Stopword Removal

Incorrect Date Range

Issue: The analysis was using a rolling 3-year window instead of a fixed start date.

Fix: Set the start date to December 1, 2021.

```

1 from datetime import datetime
2
3 start_date = datetime(2021, 12, 1)
4 end_date = datetime.utcnow()

```

Listing 6.6: Fixing Date Range

Overall Prompt Summary

| Tool | Date | Version | Prompt |
|---------|----------------|-------------------------------|---|
| ChatGPT | All-Throughout | ChatGPT 4 Turbo - ChatGPT 5.2 | Proofread this text for grammar, tone and spelling. The general tone of the thesis should be professional and academic. |
| ChatGPT | 2025-01-24 | ChatGPT 4 Turbo | Analyse Discord messages for problem-related discussions using BERTopic. |
| ChatGPT | 2025-01-24 | ChatGPT 4 Turbo | Fix error related to loading a BERTopic model without an embedding model. |
| ChatGPT | 2025-01-24 | ChatGPT 4 Turbo | Extract messages from Discord API logs stored in JSON format. |
| ChatGPT | 2025-01-24 | ChatGPT 4 Turbo | Filter messages to keep only problem-related ones using a list of keywords. |
| ChatGPT | 2025-01-24 | ChatGPT 4 Turbo | Compute response times for users replying to messages and find the most active repliers. |
| ChatGPT | 2025-01-25 | ChatGPT 4 Turbo | Load time-tracking data from CSV and visualise it using matplotlib and seaborn. |
| ChatGPT | 2025-01-25 | ChatGPT 4 Turbo | Fix KeyError: 'user' when analysing replies in Discord data. |
| ChatGPT | 2025-01-25 | ChatGPT 4 Turbo | Set a fixed start date of December 1, 2021, instead of a rolling 3-year window. |
| ChatGPT | 2025-01-25 | ChatGPT 4 Turbo | Install missing Python package for GitHub interaction in Conda. |
| ChatGPT | 2025-01-26 | ChatGPT 4 Turbo | Use stopwords from sklearn and add custom stopwords to remove unnecessary words. |
| ChatGPT | 2025-01-26 | ChatGPT 4 Turbo | Generate separate non-stacked bar charts with technology on the x-axis. |

| Tool | Date | Version | Prompt |
|-------------|-------------|-----------------|--|
| ChatGPT | 2025-01-26 | ChatGPT 4 Turbo | Generate LaTeX documentation for all prompts and their final solutions. |
| ChatGPT | 2025-01-26 | ChatGPT 4 Turbo | Simplify the LaTeX documentation by consolidating prompts and solutions into sections. |
| ChatGPT | 2025-01-26 | ChatGPT 4 Turbo | Create a table listing all prompts given in this thread with tool, date, version, and description. |

List of Figures

| | | |
|------|--|----|
| 2.1 | The \mathbb{K} Framework ¹ | 8 |
| 3.1 | A simplified KEVM verification test | 15 |
| 3.2 | A simplified ERC20 token | 16 |
| 3.3 | Parts of simpletoken-bin-runtime.k | 16 |
| 3.4 | A KEVM test | 18 |
| 3.5 | The KEVM debug tree | 19 |
| 3.6 | ACT Compilation Pipeline | 20 |
| 3.7 | A Kontrol test for SimpleToken | 29 |
| 5.1 | Structure of the KEVM test environment folder | 52 |
| 5.2 | KEVM compact proof definition | 53 |
| 5.3 | Relevant parts of KEVM proof definition | 53 |
| 5.4 | KCFG of a proof unexpectedly resulting in an EVM revert | 54 |
| 5.5 | Visualization of execution during smart contract debugging using <code>simbolik</code> | 56 |
| 5.6 | KEVM Repository Issue Trends | 62 |
| 5.7 | ACT Repository Issue Trends | 63 |
| 5.8 | Kontrol Repository Issue Trends | 64 |
| 5.9 | KEVM Repository Commit Trends | 65 |
| 5.10 | ACT Repository Commit Trends | 65 |
| 5.11 | Kontrol Repository Commit Trends | 66 |
| 5.12 | KEVM Repository Contributor Statistics | 67 |
| 5.13 | ACT Repository Contributor Statistics | 67 |
| 5.14 | Kontrol Repository Contributor Statistics | 68 |
| 5.15 | Distribution of user-engagement | 69 |
| 5.16 | Top Contributors | 70 |
| 5.17 | Message Characterization as per [Wu et al., 2024] | 72 |
| 5.18 | Message characterization as per [Wu et al., 2024] over time | 72 |

List of Tables

| | | |
|-----|--|----|
| 5.1 | Bittelux Transfer Tests | 60 |
| 5.2 | Bittelux Test Results | 60 |
| 5.3 | BitteluxUnchecked Test Results | 61 |

Bibliography

- [Amani et al., 2018] Amani, S., Bégel, M., Bortin, M., and Staples, M. (2018). Towards verifying Ethereum smart contract bytecode in isabelle/hol. CPP 2018, page 66–77, New York, NY, USA. Association for Computing Machinery.
- [Atzei et al., 2017] Atzei, N., Bartoletti, M., and Cimoli, T. (2017). A survey of attacks on Ethereum smart contracts (sok). In *International conference on principles of security and trust*, pages 164–186. Springer.
- [Buterin et al., 2014] Buterin, V. et al. (2014). Ethereum white paper. https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf.
- [Coelho et al., 2020] Coelho, J., Valente, M. T., Milen, L., and Silva, L. L. (2020). Is this github project maintained? measuring the level of maintenance activity of open-source projects. *Information and Software Technology*, 122:106274.
- [DappHub, 2019] DappHub (2019). KLab: Formal Verification for EVM Smart Contracts. <https://github.com/dapphub/klab/blob/master/README.md>. Accessed: 2025-02-28.
- [Ethereum Foundation, 2019] Ethereum Foundation (2019). The Act Specification Language. <https://ethereum.github.io/act/language.html>. Accessed: 2025-02-28.
- [Foundry, 2023] Foundry (2023). Foundry book. <https://book.getfoundry.sh/>. Accessed: 2025-03-19.
- [Grishchenko et al., 2018] Grishchenko, I., Maffei, M., and Schneidewind, C. (2018). Foundations and tools for the static analysis of Ethereum smart contracts. In *International Conference on Computer Aided Verification*, pages 51–78. Springer.
- [Hellman et al., 2022] Hellman, J., Chen, J., Uddin, M. S., Cheng, J., and Guo, J. L. C. (2022). Characterizing user behaviors in open-source software user forums: an empirical study. In *Proceedings of the 15th International Conference on Cooperative and Human Aspects of Software Engineering, ICSE '22*, page 46–55. ACM.

- [Hildenbrandt et al., 2018] Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., and Roşu, G. (2018). Kevm: A complete semantics of the Ethereum virtual machine. Technical report.
- [Hirai, 2017] Hirai, Y. (2017). Defining the Ethereum Virtual Machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*, pages 520–535. Springer.
- [Kinariwala and Deshmukh, 2023] Kinariwala, S. and Deshmukh, S. (2023). Short text topic modelling using local and global word-context semantic correlation. *Multimedia Tools and Applications*, pages 1–23. Advance online publication.
- [Luu et al., 2016] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. (2016). Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269.
- [Rameder et al., 2022] Rameder, H., Di Angelo, M., and Salzer, G. (2022). Review of automated vulnerability analysis of smart contracts on Ethereum. *Front. Blockchain*, 5.
- [Rosu, 2017] Rosu, G. (2017). K: A semantic framework for programming languages and formal analysis tools. In *Dependable Software Systems Engineering*, pages 186–206. IOS Press.
- [Runtime Verification, 2023a] Runtime Verification, I. (2023a). Kontrol documentation. <https://docs.runtimeverification.com/kontrol>. Accessed: 2025-03-19.
- [Runtime Verification, 2023b] Runtime Verification, I. (2023b). Kontrol github. <https://github.com/runtimeverification/kontrol/>. Accessed: 2025-03-19.
- [Schneidewind et al., 2020] Schneidewind, C., Grishchenko, I., Scherer, M., and Maffei, M. (2020). ethor: Practical and provably sound static analysis of Ethereum smart contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 621–640.
- [Sooho.io, 2023] Sooho.io, J. P. (2023). Verismartbench: CVE false reported cases. <https://github.com/soohoio/VeriSmartBench/wiki/CVE-False-Reported-Case>. Accessed: February 17, 2026.
- [Wood et al., 2014] Wood, G. et al. (2014). Ethereum: A secure decentralised generalised transaction ledger. <https://ethereum.github.io/yellowpaper/paper.pdf>.
- [Wu et al., 2024] Wu, X., Laufer, E., Li, H., Khomh, F., Srinivasan, S., and Luo, J. (2024). Characterizing and classifying developer forum posts with their intentions. *Empirical Software Engineering*, 29(4):84.

Addendum

Code and relevant artefacts created for this thesis are also available under:
https://github.com/splessberger/thesis_kevm_praxis/.

A.1 Smart Contract Code

A.1.1 HelloWorld.sol

```
1 contract HelloWorld {
2     function sayHelloWorld() public pure returns (string memory) {
3         return "Hello World";
4     }
5 }
```

A.1.2 SimpleToken.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 contract SimpleToken {
5     mapping(address => uint256) private _balances;
6     uint256 private _totalSupply;
7     string private _name;
8     string private _symbol;
9     address private _owner;
10
11     constructor() {
12         _name = 'SimpleToken';
13         _symbol = 'ST';
14         _owner = msg.sender;
15     }
16
17     function name() external view returns (string memory) {
18         return _name;
19     }
20
21     function symbol() external view returns (string memory) {
22         return _symbol;
23     }
24
25     function decimals() external pure returns (uint8) {
26         return 0;
27     }
28 }
```

```

29     function totalSupply() external view returns (uint256) {
30         return _totalSupply;
31     }
32
33     function standard() external pure returns (string memory) {
34         return 'erc20';
35     }
36
37     function balanceOf(address account) external view returns (uint256) {
38         return _balances[account];
39     }
40
41     function transfer(address to, uint256 value) external returns (bool) {
42         _balances[msg.sender] = _balances[msg.sender] - value;
43         _balances[to] = _balances[to] + value;
44         return true;
45     }
46
47     function mint(address account, uint256 amount) external returns (bool) {
48         if(_owner != msg.sender) revert();
49         require(account != address(0), "ERC20: mint to the zero address");
50
51         _totalSupply += amount;
52         _balances[account] += amount;
53         return true;
54     }
55
56     function burn(uint256 amount) external {
57         uint256 accountBalance = _balances[msg.sender];
58         require(accountBalance >= amount, "ERC20: burn amount exceeds balance");
59         _balances[msg.sender] = accountBalance - amount;
60         _totalSupply -= amount;
61     }
62 }

```

A.1.3 StandardToken.sol

```

1  pragma solidity ^0.8.13;
2
3  contract Token {
4      function totalSupply() external virtual returns (uint256 supply) {}
5      function balanceOf(address _owner) public virtual returns (uint256 balance) {}
6      function transfer(address _to, uint256 _value) public virtual returns (bool success)
7          {}
8      function transferFrom(address _from, address _to, uint256 _value) public virtual
9          returns (bool success) {}
10     function approve(address _spender, uint256 _value) public virtual returns (bool
11         success) {}
12     function allowance(address _owner, address _spender) public virtual returns (uint256
13         remaining) {}
14     event Transfer(address indexed _from, address indexed _to, uint256 _value);
15     event Approval(address indexed _owner, address indexed _spender, uint256 _value);
16 }
17
18 contract StandardToken is Token {
19     function transfer(address _to, uint256 _value) public override returns (bool success)
20     {
21         if (balances[msg.sender] >= _value && _value > 0) {
22             balances[msg.sender] -= _value;
23             balances[_to] += _value;

```

```

19     emit Transfer(msg.sender, _to, _value);
20     return true;
21 } else { return false; }
22 }
23
24 function transferFrom(address _from, address _to, uint256 _value) public override
    returns (bool success) {
25     if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && _value >
        0) {
26         balances[_to] += _value;
27         balances[_from] -= _value;
28         allowed[_from][msg.sender] -= _value;
29         emit Transfer(_from, _to, _value);
30         return true;
31     } else { return false; }
32 }
33
34 function balanceOf(address _owner) public override returns (uint256 balance) {
35     return balances[_owner];
36 }
37
38 function approve(address _spender, uint256 _value) public override returns (bool
    success) {
39     allowed[msg.sender][_spender] = _value;
40     emit Approval(msg.sender, _spender, _value);
41     return true;
42 }
43
44 function allowance(address _owner, address _spender) public override returns (uint256
    remaining) {
45     return allowed[_owner][_spender];
46 }
47
48 mapping (address => uint256) balances;
49 mapping (address => mapping (address => uint256)) allowed;
50 uint256 public override totalSupply;
51 }

```

A.1.4 StandardTokenUnchecked.sol

```

1 pragma solidity ^0.8.13;
2
3 contract Token {
4     function totalSupply() external virtual returns (uint256 supply) {}
5     function balanceOf(address _owner) public virtual returns (uint256 balance) {}
6     function transfer(address _to, uint256 _value) public virtual returns (bool success)
    {}
7     function transferFrom(address _from, address _to, uint256 _value) public virtual
    returns (bool success) {}
8     function approve(address _spender, uint256 _value) public virtual returns (bool
    success) {}
9     function allowance(address _owner, address _spender) public virtual returns (uint256
    remaining) {}
10    event Transfer(address indexed _from, address indexed _to, uint256 _value);
11    event Approval(address indexed _owner, address indexed _spender, uint256 _value);
12 }
13
14 contract StandardTokenUnchecked is Token {
15     function transfer(address _to, uint256 _value) public override returns (bool success)
    {}

```

```

16     if (balances[msg.sender] >= _value && _value > 0) {
17         unchecked {
18             balances[msg.sender] -= _value;
19             balances[_to] += _value;
20         }
21         emit Transfer(msg.sender, _to, _value);
22         return true;
23     } else { return false; }
24 }
25
26 function transferFrom(address _from, address _to, uint256 _value) public override
    returns (bool success) {
27     if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && _value >
        0) {
28         unchecked {
29             balances[_to] += _value;
30             balances[_from] -= _value;
31             allowed[_from][msg.sender] -= _value;
32         }
33         emit Transfer(_from, _to, _value);
34         return true;
35     } else { return false; }
36 }
37
38 function balanceOf(address _owner) public override returns (uint256 balance) {
39     return balances[_owner];
40 }
41
42 function approve(address _spender, uint256 _value) public override returns (bool
    success) {
43     allowed[msg.sender][_spender] = _value;
44     emit Approval(msg.sender, _spender, _value);
45     return true;
46 }
47
48 function allowance(address _owner, address _spender) public override returns (uint256
    remaining) {
49     return allowed[_owner][_spender];
50 }
51
52 mapping (address => uint256) balances;
53 mapping (address => mapping (address => uint256)) allowed;
54 uint256 public override totalSupply;
55 }

```

A.1.5 Bittelux.sol

```

1 pragma solidity ^0.8.13;
2
3 import "./StandardToken.sol";
4
5 contract Bittelux is StandardToken {
6     string public name;
7     uint8 public decimals;
8     string public symbol;
9     string public version = 'H1.0';
10    uint256 public unitsOneEthCanBuy;
11    uint256 public totalEthInWei;
12    address payable public fundsWallet;
13 }

```



```

23
24     receive() external payable {
25         totalEthInWei = totalEthInWei + msg.value;
26         uint256 amount = msg.value * unitsOneEthCanBuy;
27         require(balances[fundsWallet] >= amount);
28
29         unchecked {
30             balances[fundsWallet] = balances[fundsWallet] - amount;
31             balances[msg.sender] = balances[msg.sender] + amount;
32         }
33
34         emit Transfer(fundsWallet, msg.sender, amount);
35
36         fundsWallet.transfer(msg.value);
37     }
38
39     function approveAndCall(address _spender, uint256 _value, bytes memory _extraData)
40     external returns (bool success) {
41         allowed[msg.sender][_spender] = _value;
42         emit Approval(msg.sender, _spender, _value);
43
44         (bool success, bytes memory data) = _spender.call(abi.encodeWithSignature("
45             receiveApproval(address,uint256,address,bytes)", msg.sender, _value, this,
46             _extraData));
47         if(!success) { revert(); }
48         return true;
49     }

```

A.1.7 CCLAG.sol

```

1  pragma solidity ^0.8.13;
2
3  import "./StandardToken.sol";
4
5  contract ChuCunLingAIGO is StandardToken {
6      fallback() external {
7          revert();
8      }
9
10     string public name;
11     uint8 public decimals;
12     string public symbol;
13     string public version = 'H0.1';
14
15     constructor(
16         uint256 _initialAmount,
17         string memory _tokenName,
18         uint8 _decimalUnits,
19         string memory _tokenSymbol
20     ) {
21         balances[msg.sender] = _initialAmount;
22         totalSupply = _initialAmount;
23         name = _tokenName;
24         decimals = _decimalUnits;
25         symbol = _tokenSymbol;
26     }
27
28     function approveAndCall(address _spender, uint256 _value, bytes memory _extraData)
29     public returns (bool success) {

```

```

29     allowed[msg.sender][_spender] = _value;
30     emit Approval(msg.sender, _spender, _value);
31     if(!_spender.call(bytes4(bytes32(sha3("receiveApproval(address,uint256,address,
32         bytes)"))), msg.sender, _value, this, _extraData)) { throw; }
33     return true;
34 }

```

A.1.8 CCLAGUnchecked.sol

```

1  pragma solidity ^0.8.13;
2
3  import "./StandardTokenUnchecked.sol";
4
5  contract ChuCunLingAIGOUUnchecked is StandardTokenUnchecked {
6
7      fallback() external {
8          revert();
9      }
10
11     string public name;
12     uint8 public decimals;
13     string public symbol;
14     string public version = 'H0.1';
15
16     constructor(
17         uint256 _initialAmount,
18         string memory _tokenName,
19         uint8 _decimalUnits,
20         string memory _tokenSymbol
21     ) {
22         balances[msg.sender] = _initialAmount;
23         totalSupply = _initialAmount;
24         name = _tokenName;
25         decimals = _decimalUnits;
26         symbol = _tokenSymbol;
27     }
28
29     function approveAndCall(address _spender, uint256 _value, bytes memory _extraData)
30         public returns (bool success) {
31         allowed[msg.sender][_spender] = _value;
32         emit Approval(msg.sender, _spender, _value);
33         if(!_spender.call(bytes4(bytes32(sha3("receiveApproval(address,uint256,address,
34             bytes)"))), msg.sender, _value, this, _extraData)) { throw; }
35         return true;
36     }
37 }

```

A.1.9 ETT.sol

```

1  pragma solidity ^0.8.13;
2
3  import "./StandardToken.sol";
4
5  contract HashnodeTestCoin is StandardToken {
6     string public name;
7     uint8 public decimals;
8     string public symbol;
9     string public version = 'H1.0';

```



```

18     decimals = 18;
19     symbol = "ETT";
20     unitsOneEthCanBuy = 40000;
21     fundsWallet = payable(address(msg.sender));
22 }
23
24 receive() external payable {
25     totalEthInWei = totalEthInWei + msg.value;
26     uint256 amount = msg.value * unitsOneEthCanBuy;
27     require(balances[fundsWallet] >= amount);
28
29     unchecked {
30         balances[fundsWallet] = balances[fundsWallet] - amount;
31         balances[msg.sender] = balances[msg.sender] + amount;
32     }
33
34     emit Transfer(fundsWallet, msg.sender, amount); // Broadcast a message to the
        blockchain
35     fundsWallet.transfer(msg.value);
36 }
37
38 function approveAndCall(address _spender, uint256 _value, bytes memory _extraData)
    public returns (bool success) {
39     allowed[msg.sender][_spender] = _value;
40     emit Approval(msg.sender, _spender, _value);
41
42     (bool success, bytes memory data) = _spender.call(abi.encodeWithSignature("
        receiveApproval(address,uint256,address,bytes)", msg.sender, _value, this,
        _extraData));
43     if(!success) { revert(); }
44     return true;
45 }
46 }

```

A.1.11 Pandora.sol

```

1  pragma solidity ^0.8.13;
2
3  import "./StandardToken.sol";
4
5  contract HumanStandardToken is StandardToken {
6     string public name;
7     uint8 public decimals;
8     string public symbol;
9     string public version = 'H0.1';
10
11     constructor(uint256 _initialAmount, string memory _tokenName, uint8 _decimalUnits,
        string memory _tokenSymbol) {
12         balances[msg.sender] = _initialAmount;
13         totalSupply = _initialAmount;
14         name = _tokenName;
15         decimals = _decimalUnits;
16         symbol = _tokenSymbol;
17     }
18
19     function approveAndCall(address _spender, uint256 _value, bytes memory _extraData)
        public returns (bool success) {
20         allowed[msg.sender][_spender] = _value;
21         emit Approval(msg.sender, _spender, _value);

```

```

22     (bool success, bytes memory data) = _spender.call(abi.encodeWithSignature("
        receiveApproval(address,uint256,address,bytes)", msg.sender, _value, this,
        _extraData));
23     if(!success) { revert(); }
24     return true;
25 }
26 }

```

A.1.12 PandoraUnchecked.sol

```

1  pragma solidity ^0.8.13;
2
3  import "./StandardTokenUnchecked.sol";
4
5  contract HumanStandardTokenUnchecked is StandardTokenUnchecked {
6      string public name;
7      uint8 public decimals;
8      string public symbol;
9      string public version = 'H0.1';
10
11     constructor(uint256 _initialAmount, string memory _tokenName, uint8 _decimalUnits,
12         string memory _tokenSymbol) {
13         balances[msg.sender] = _initialAmount;
14         totalSupply = _initialAmount;
15         name = _tokenName;
16         decimals = _decimalUnits;
17         symbol = _tokenSymbol;
18     }
19
20     function approveAndCall(address _spender, uint256 _value, bytes memory _extraData)
21         public returns (bool success) {
22         allowed[msg.sender][_spender] = _value;
23         emit Approval(msg.sender, _spender, _value);
24         (bool success, bytes memory data) = _spender.call(abi.encodeWithSignature("
25             receiveApproval(address,uint256,address,bytes)", msg.sender, _value, this,
26             _extraData));
27         if(!success) { revert(); }
28         return true;
29     }
30 }

```

A.2 KEVM Test Specifications

A.2.1 simpletoken-runtime.k

```

1  requires "eds1.md"
2
3  module SIMPLETOKEN-CONTRACT
4      imports public EDSL
5
6      syntax Contract ::= SimpleTokenContract
7      syntax SimpleTokenContract ::= "SimpleToken" [symbol(), klable(contract_SimpleToken
8          )]
9
10     rule ( #binRuntime ( SimpleToken ) => #parseByteStack ( "0x..." ) )
11
12     syntax Field ::= SimpleTokenField

```

```

12  syntax SimpleTokenField ::= "_balances" [symbol(), klabel(
      field_SimpleToken_balances)]
13  syntax SimpleTokenField ::= "_totalSupply" [symbol(), klabel(
      field_SimpleToken__totalSupply)]
14  syntax SimpleTokenField ::= "_name" [symbol(), klabel(field_SimpleToken__name)]
15  syntax SimpleTokenField ::= "_symbol" [symbol(), klabel(field_SimpleToken__symbol)]
16  syntax SimpleTokenField ::= "_owner" [symbol(), klabel(field_SimpleToken__owner)]
17
18  rule ( #loc ( SimpleToken . _balances ) => 0 )
19  rule ( #loc ( SimpleToken . _totalSupply ) => 1 )
20  rule ( #loc ( SimpleToken . _name ) => 2 )
21  rule ( #loc ( SimpleToken . _symbol ) => 3 )
22  rule ( #loc ( SimpleToken . _owner ) => 4 )
23
24  syntax Bytes ::= SimpleTokenContract "." SimpleTokenMethod [function(), symbol(),
      klabel(method_SimpleToken)]
25  syntax SimpleTokenMethod ::= "balanceOf" "(" Int ":" "address" ")" [symbol(),
      klabel(method_SimpleToken_balanceOf_address)]
26  syntax SimpleTokenMethod ::= "burn" "(" Int ":" "uint256" ")" [symbol(), klabel(
      method_SimpleToken_burn_uint256)]
27  syntax SimpleTokenMethod ::= "decimals" "(" ")" [symbol(), klabel(
      method_SimpleToken_decimals_)]
28  syntax SimpleTokenMethod ::= "mint" "(" Int ":" "address" "," Int ":" "uint256" ")"
      [symbol(), klabel(method_SimpleToken_mint_address_uint256)]
29  syntax SimpleTokenMethod ::= "name" "(" ")" [symbol(), klabel(
      method_SimpleToken_name_)]
30  syntax SimpleTokenMethod ::= "standard" "(" ")" [symbol(), klabel(
      method_SimpleToken_standard_)]
31  syntax SimpleTokenMethod ::= "symbol" "(" ")" [symbol(), klabel(
      method_SimpleToken_symbol_)]
32  syntax SimpleTokenMethod ::= "totalSupply" "(" ")" [symbol(), klabel(
      method_SimpleToken_totalSupply_)]
33  syntax SimpleTokenMethod ::= "transfer" "(" Int ":" "address" "," Int ":" "uint256"
      ")" [symbol(), klabel(method_SimpleToken_transfer_address_uint256)]
34
35  rule ( SimpleToken . balanceOf ( V0_account : address ) => #abiCallData ( "
      balanceOf" , #address ( V0_account ) , .TypedArgs ) )
36  ensures #rangeAddress ( V0_account )
37
38  rule ( SimpleToken . burn ( V0_amount : uint256 ) => #abiCallData ( "burn" , #
      uint256 ( V0_amount ) , .TypedArgs ) )
39  ensures #rangeUInt ( 256 , V0_amount )
40
41  rule ( SimpleToken . decimals ( ) => #abiCallData ( "decimals" , .TypedArgs ) )
42
43  rule ( SimpleToken . mint ( V0_account : address , V1_amount : uint256 ) => #
      abiCallData ( "mint" , #address ( V0_account ) , #uint256 ( V1_amount ) , .
      TypedArgs ) )
44  ensures ( #rangeAddress ( V0_account )
45  andBool ( #rangeUInt ( 256 , V1_amount )
46  ) )
47
48  rule ( SimpleToken . name ( ) => #abiCallData ( "name" , .TypedArgs ) )
49  rule ( SimpleToken . standard ( ) => #abiCallData ( "standard" , .TypedArgs ) )
50  rule ( SimpleToken . symbol ( ) => #abiCallData ( "symbol" , .TypedArgs ) )
51  rule ( SimpleToken . totalSupply ( ) => #abiCallData ( "totalSupply" , .TypedArgs
      ) )
52
53  rule ( SimpleToken . transfer ( V0_to : address , V1_value : uint256 ) => #
      abiCallData ( "transfer" , #address ( V0_to ) , #uint256 ( V1_value ) , .
      TypedArgs ) )
54  ensures ( #rangeAddress ( V0_to )
  
```

```

55     andBool ( #rangeUInt ( 256 , V1_value )
56             )
57
58     rule ( selector ( "balanceOf(address)" ) => 1889567281 )
59     rule ( selector ( "burn(uint256)" ) => 1117154408 )
60     rule ( selector ( "decimals()" ) => 826074471 )
61     rule ( selector ( "mint(address,uint256)" ) => 1086394137 )
62     rule ( selector ( "name()" ) => 117300739 )
63     rule ( selector ( "standard()" ) => 1513848386 )
64     rule ( selector ( "symbol()" ) => 2514000705 )
65     rule ( selector ( "totalSupply()" ) => 404098525 )
66     rule ( selector ( "transfer(address,uint256)" ) => 2835717307 )
67
68
69 endmodule
70
71 module MAIN
72     imports public SIMPLETOKEN-CONTRACT
73
74
75
76 endmodule

```

A.2.2 simpletoken-verification-spec.k

```

1 requires "edsl.md"
2 requires "optimizations.md"
3 requires "lemmas/lemmas.k"
4 requires "../bin/simpletoken-runtime.k"
5
6 module SIMPLETOKEN-VERIFICATION
7     imports EDSL
8     imports LEMMAS
9     imports EVM-OPTIMIZATIONS
10    imports SIMPLETOKEN-MAIN
11 endmodule

```

A.2.3 simpletoken-mint-success-spec.k

```

1 requires "../simpletoken-verification-spec.k"
2
3 module SIMPLETOKEN-TRANSFER-SUCCESS-SPEC
4     imports SIMPLETOKEN-VERIFICATION
5
6     claim [transfer.success]:
7         <mode>      NORMAL      </mode>
8         <schedule> CONSTANTINOPLE </schedule>
9
10        <callStack> .List </callStack>
11        <program>   #binRuntime(SimpleToken) </program>
12        <jumpDests> #computeValidJumpDests(#binRuntime(SimpleToken)) </jumpDests>
13        <static>    false </static>
14
15        <id>         ACCTID      => ?_ </id>
16        <caller>    OWNER       => ?_ </caller>
17        <localMem>  .Bytes       => ?_ </localMem>
18        <memoryUsed> 0          => ?_ </memoryUsed>
19        <wordStack> .WordStack  => ?_ </wordStack>
20        <pc>        0           => ?_ </pc>

```

```

21   <gas>      #gas(_VGAS) => ?_ </gas>
22   <callValue> 0          => ?_ </callValue>
23
24   <callData> SimpleToken . transfer ( V0_to : address, V1_value : uint256 )
                </callData>
25   <k>        #execute => #halt ...          </k>
26   <output>   .Bytes    => #buf(32, 1) </output>
27   <statusCode> _      => EVMC_SUCCESS    </statusCode>
28
29   <substate>
30     <selfDestruct> _ </selfDestruct>
31     <log> _ => ?_ </log>
32     <refund> _ => ?_ </refund>
33     <accessedAccounts> _ => ?_ </accessedAccounts>
34     <accessedStorage> _ => ?_ </accessedStorage>
35   </substate>
36
37   <account>
38     <acctID> ACCTID </acctID>
39     <code> #binRuntime(SimpleToken) </code>
40     <storage>
41       #hashedLocation("Solidity", 0, V0_to) |-> (BAL_TO => BAL_TO +Int V1_value)
42       #hashedLocation("Solidity", 0, OWNER) |-> (BAL_FROM => BAL_FROM -Int V1_value)
43     </storage>
44     <origStorage>
45       #hashedLocation("Solidity", 0, V0_to) |-> BAL_TO
46       #hashedLocation("Solidity", 0, OWNER) |-> BAL_FROM
47     </origStorage>
48     ...
49   </account>
50
51   requires #rangeAddress(V0_to)
52   andBool #rangeAddress(ACCTID)
53   andBool #rangeAddress(OWNER)
54   andBool #rangeUInt(256, V1_value)
55   andBool #rangeUInt(256, BAL_TO)
56   andBool #rangeUInt(256, BAL_FROM)
57   andBool V1_value <Int BAL_FROM
58   andBool V1_value <Int 0
59   andBool ACCTID <Int 0
60   andBool OWNER <Int 0
61   andBool V0_to <Int 0
62   andBool V0_to <Int OWNER
63   andBool V0_to <Int ACCTID
64   andBool (BAL_TO +Int V1_value) <Int (2 ^Int 256)
65   andBool (BAL_FROM -Int V1_value) >Int 0
66 endmodule

```

A.2.4 simpletoken-transfer-success-spec.k

```

1 requires "./simpletoken-verification-spec.k"
2
3 module SIMPLETOKEN-TRANSFER-SUCCESS-SPEC
4   imports SIMPLETOKEN-VERIFICATION
5
6   claim [transfer.success]:
7     <mode>      NORMAL    </mode>
8     <schedule>  CONSTANTINOPLE </schedule>
9
10    <callStack> .List </callStack>

```

```

11 <program> #binRuntime(SimpleToken) </program>
12 <jumpDests> #computeValidJumpDests(#binRuntime(SimpleToken)) </jumpDests>
13 <static> false </static>
14
15 <id> ACCTID => ?_ </id>
16 <caller> OWNER => ?_ </caller>
17 <localMem> .Bytes => ?_ </localMem>
18 <memoryUsed> 0 => ?_ </memoryUsed>
19 <wordStack> .WordStack => ?_ </wordStack>
20 <pc> 0 => ?_ </pc>
21 <gas> #gas(_VGAS) => ?_ </gas>
22 <callValue> 0 => ?_ </callValue>
23
24 <callData> SimpleToken . transfer ( V0_to : address, V1_value : uint256 )
    </callData>
25 <k> #execute => #halt ... </k>
26 <output> .Bytes => #buf(32, 1) </output>
27 <statusCode> _ => EVMC_SUCCESS </statusCode>
28
29 <substate>
30 <selfDestruct> _ </selfDestruct>
31 <log> _ => ?_ </log>
32 <refund> _ => ?_ </refund>
33 <accessedAccounts> _ => ?_ </accessedAccounts>
34 <accessedStorage> _ => ?_ </accessedStorage>
35 </substate>
36
37 <account>
38 <acctID> ACCTID </acctID>
39 <code> #binRuntime(SimpleToken) </code>
40 <storage>
41 #hashedLocation("Solidity", 0, V0_to) |-> (BAL_TO => BAL_TO +Int V1_value)
42 #hashedLocation("Solidity", 0, OWNER) |-> (BAL_FROM => BAL_FROM -Int V1_value)
43 </storage>
44 <origStorage>
45 #hashedLocation("Solidity", 0, V0_to) |-> BAL_TO
46 #hashedLocation("Solidity", 0, OWNER) |-> BAL_FROM
47 </origStorage>
48 ...
49 </account>
50
51 requires #rangeAddress(V0_to)
52 andBool #rangeAddress(ACCTID)
53 andBool #rangeAddress(OWNER)
54 andBool #rangeUInt(256, V1_value)
55 andBool #rangeUInt(256, BAL_TO)
56 andBool #rangeUInt(256, BAL_FROM)
57 andBool V1_value <Int BAL_FROM
58 andBool V1_value !=Int 0
59 andBool ACCTID !=Int 0
60 andBool OWNER !=Int 0
61 andBool V0_to !=Int 0
62 andBool V0_to !=Int OWNER
63 andBool V0_to !=Int ACCTID
64 andBool (BAL_TO +Int V1_value) <Int (2 ^Int 256)
65 andBool (BAL_FROM -Int V1_value) >=Int 0
66
67 endmodule

```

A.2.5 /bittelux-transfer-success-spec.k


```

60     andBool (BAL_TO +Int BAL_FROM) <=Int TOTAL_SUPPLY
61     andBool TOTAL_SUPPLY <Int (2 ^Int 256)
62 endmodule

```

A.2.6 cclag-transfer-success-spec.k

```

1  requires "./cclag-runtime.k"
2
3  module CCLAG-TRANSFER-SUCCESS-SPEC
4      imports CCLAG-VERIFICATION
5
6      claim [transfer.success]:
7          <mode>      NORMAL </mode>
8          <schedule> CONSTANTINOPIE </schedule>
9
10         <callStack> .List </callStack>
11         <program>   #binRuntime(ChuCunLingAIGO) </program>
12         <jumpDests> #computeValidJumpDests(#binRuntime(ChuCunLingAIGO)) </jumpDests>
13         <static>    false </static>
14
15         <id>         ACCTID      => ?_ </id>
16         <caller>    OWNER        => ?_ </caller>
17         <localMem>  .Bytes       => ?_ </localMem>
18         <memoryUsed> 0           => ?_ </memoryUsed>
19         <wordStack> .WordStack  => ?_ </wordStack>
20         <pc>        0            => ?_ </pc>
21         <gas>       #gas(_VGAS) => ?_ </gas>
22         <callValue> 0           => ?_ </callValue>
23
24         <callData> ChuCunLingAIGO . transfer ( V0__to : address , V1__value : uint256 ) </
                callData>
25         <k>         #execute => #halt ...           </k>
26         <output>    .Bytes    => ?_ </output>
27         <statusCode> _        => EVMC_SUCCESS     </statusCode>
28
29         <substate>
30             <selfDestruct> _ </selfDestruct>
31             <log> _ => ?_ </log>
32             <refund> _ => ?_ </refund>
33             <accessedAccounts> _ => ?_ </accessedAccounts>
34             <createdAccounts> _ => ?_ </createdAccounts>
35             <accessedStorage> _ => ?_ </accessedStorage>
36         </substate>
37
38         <account>
39             <acctID> ACCTID </acctID>
40             <code> #binRuntime(ChuCunLingAIGO) </code>
41             <storage> (#hashedLocation("Solidity", 0, V0__to) |-> (BAL_TO => BAL_TO +Int
                V1__value)) (#hashedLocation("Solidity", 0, OWNER) |-> (BAL_FROM => BAL_FROM -
                Int V1__value)) ACCT_STORAGE:Map </storage>
42             ...
43         </account>
44
45     requires #rangeAddress(ACCTID)
46     andBool #rangeAddress(OWNER)
47     andBool #rangeUInt(256, BAL_TO)
48     andBool #rangeUInt(256, BAL_FROM)
49     andBool TOTAL_SUPPLY_LOCATION ==Int #loc(ChuCunLingAIGO.totalSupply)
50     andBool TOTAL_SUPPLY ==Int #lookup(ACCT_STORAGE, TOTAL_SUPPLY_LOCATION)
51     andBool OWNER /=Int 0

```

```

52 andBool ACCTID !=Int 0
53 andBool V0__to !=Int 0
54 andBool V0__to !=Int ACCTID
55 andBool V0__to !=Int OWNER
56 andBool ACCTID !=Int OWNER
57 andBool 0 <Int (BAL_TO +Int V1__value)
58 andBool (BAL_TO +Int V1__value) <=Int (BAL_TO +Int BAL_FROM)
59 andBool (BAL_TO +Int BAL_FROM) <=Int TOTAL_SUPPLY
60 andBool TOTAL_SUPPLY <Int (2 ^Int 256)
61 endmodule

```

A.2.7 ett-transfer-success-spec.k

```

1  requires "./ett-runtime.k"
2
3  module ETT-TRANSFER-SUCCESS-SPEC
4    imports ETT-VERIFICATION
5
6    claim [transfer.success]:
7      <mode>      NORMAL </mode>
8      <schedule> CONSTANTINOPIE </schedule>
9
10     <callStack> .List </callStack>
11     <program>   #binRuntime(HashnodeTestCoin) </program>
12     <jumpDests> #computeValidJumpDests(#binRuntime(HashnodeTestCoin)) </jumpDests>
13     <static>    false </static>
14
15     <id>        ACCTID      => ?_ </id>
16     <caller>    OWNER       => ?_ </caller>
17     <localMem>  .Bytes      => ?_ </localMem>
18     <memoryUsed> 0          => ?_ </memoryUsed>
19     <wordStack> .WordStack => ?_ </wordStack>
20     <pc>        0           => ?_ </pc>
21     <gas>       #gas(_VGAS) => ?_ </gas>
22     <callValue> 0           => ?_ </callValue>
23
24     <callData> HashnodeTestCoin . transfer ( V0__to : address , V1__value : uint256 ) <
25       /callData>
26     <k>         #execute => #halt ...           </k>
27     <output>    .Bytes   => ?_ </output>
28     <statusCode> _      => EVMC_SUCCESS       </statusCode>
29
30     <substate>
31       <selfDestruct> _ </selfDestruct>
32       <log> _ => ?_ </log>
33       <refund> _ => ?_ </refund>
34       <accessedAccounts> _ => ?_ </accessedAccounts>
35       <accessedStorage> _ => ?_ </accessedStorage>
36       <createdAccounts> _ => ?_ </createdAccounts>
37     </substate>
38
39     <account>
40       <acctID> ACCTID </acctID>
41       <code> #binRuntime(HashnodeTestCoin) </code>
42       <storage> (#hashedLocation("Solidity", 0, V0__to) |-> (BAL_TO => BAL_TO +Int
43         V1__value)) (#hashedLocation("Solidity", 0, OWNER) |-> (BAL_FROM => BAL_FROM -
44         Int V1__value)) ACCT_STORAGE:Map </storage>
45     ...
46   </account>
47
48

```



```

39 <acctID> ACCTID </acctID>
40 <code> #binRuntime(HumanStandardToken) </code>
41 <storage> (#hashedLocation("Solidity", 0, V0__to) |-> (BAL_TO => BAL_TO +Int
      V1__value)) (#hashedLocation("Solidity", 0, OWNER) |-> (BAL_FROM => BAL_FROM -
      Int V1__value)) ACCT_STORAGE:Map </storage>
42 ...
43 </account>
44
45 requires #rangeAddress(ACCTID)
46 andBool #rangeAddress(OWNER)
47 andBool #rangeUInt(256, BAL_TO)
48 andBool #rangeUInt(256, BAL_FROM)
49 andBool TOTAL_SUPPLY_LOCATION ==Int #loc(HumanStandardToken.totalSupply)
50 andBool TOTAL_SUPPLY ==Int #lookup(ACCT_STORAGE, TOTAL_SUPPLY_LOCATION)
51 andBool OWNER !=Int 0
52 andBool ACCTID !=Int 0
53 andBool V0__to !=Int 0
54 andBool V0__to !=Int ACCTID
55 andBool V0__to !=Int OWNER
56 andBool ACCTID !=Int OWNER
57 andBool 0 <Int (BAL_TO +Int V1__value)
58 andBool (BAL_TO +Int V1__value) <=Int (BAL_TO +Int BAL_FROM)
59 andBool (BAL_TO +Int BAL_FROM) <=Int TOTAL_SUPPLY
60 andBool TOTAL_SUPPLY <Int (2 ^Int 256)
61 endmodule

```

A.3 Kontrol Test Specifications

A.3.1 BTXFull.t.sol

```

1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.13;
3
4 import "../src/Bittelux.sol";
5 import "forge-std/Test.sol";
6 import "kontrol-cheatcodes/KontrolCheats.sol";
7
8 contract BitteluxTest is Test, KontrolCheats {
9   uint256 constant MAX_INT = 2**256 - 1;
10  Bittelux token; // Contract under test
11
12  function setUp() public {
13    token = new Bittelux();
14  }
15
16  function testEqualReceive(address alice, address ceasar, uint256 amount) public {
17    kevm.infiniteGas();
18
19    vm.assume(alice != address(0));
20    vm.assume(ceasar != address(0));
21    vm.assume(alice == token.fundsWallet());
22    vm.assume(notBuiltinAddress(alice));
23    vm.assume(notBuiltinAddress(ceasar));
24
25    vm.assume(token.balanceOf(alice) <= token.totalSupply());
26
27    uint256 oldBalanceAlice = token.balanceOf(alice);
28    uint256 oldBalanceCeasar = token.balanceOf(ceasar);
29    uint256 oldTotalSupply = token.totalSupply();

```

```

30
31 vm.prank(alice);
32 (bool sent, bytes memory data) = payable(token).call{value: amount}("");
33
34 assertEquals(token.balanceOf(alice), oldBalanceAlice);
35 assertEquals(token.balanceOf(ceasar), oldBalanceCeasar);
36
37 assertEquals(oldTotalSupply, token.totalSupply());
38
39 assertLe(token.balanceOf(alice), token.totalSupply());
40 }
41
42 function testReceive(address alice, address ceasar, uint256 amount) public {
43     kevm.infiniteGas();
44
45     vm.assume(alice != address(0));
46     vm.assume(ceasar != address(0));
47     vm.assume(token.fundsWallet() != address(0));
48     vm.assume(notBuiltinAddress(alice));
49     vm.assume(alice != token.fundsWallet());
50     vm.assume(notBuiltinAddress(token.fundsWallet()));
51
52     uint256 tokens_bought = amount * token.unitsOneEthCanBuy();
53
54     vm.assume((tokens_bought + token.balanceOf(alice) <= token.balanceOf(token.
55         fundsWallet()) + token.balanceOf(alice)));
56     vm.assume(token.balanceOf(token.fundsWallet()) + token.balanceOf(alice) <= token.
57         totalSupply());
58
59     uint256 oldBalanceAlice = token.balanceOf(alice);
60     uint256 oldBalanceFundsWallet = token.balanceOf(token.fundsWallet());
61     uint256 oldBalanceCeasar = token.balanceOf(ceasar);
62     uint256 oldTotalSupply = token.totalSupply();
63
64     vm.prank(alice);
65     (bool sent, bytes memory data) = payable(token).call{value: amount}("");
66
67     assertEquals(token.balanceOf(alice), oldBalanceAlice + tokens_bought);
68     assertEquals(token.balanceOf(token.fundsWallet()), oldBalanceFundsWallet - tokens_bought);
69
70     assertEquals(token.balanceOf(ceasar), oldBalanceCeasar);
71
72     assertEquals(token.balanceOf(alice) + token.balanceOf(token.fundsWallet()),
73         oldBalanceAlice + oldBalanceFundsWallet);
74     assertEquals(oldTotalSupply, token.totalSupply());
75
76     assertLe((token.balanceOf(alice) + token.balanceOf(token.fundsWallet())), token.
77         totalSupply());
78 }
79
80 function testEqualTransfer(address alice, address ceasar, uint256 amount) public {
81     kevm.infiniteGas();
82
83     vm.assume(alice != address(0));
84     vm.assume(ceasar != address(0));
85
86     vm.assume(notBuiltinAddress(alice));
87     vm.assume(notBuiltinAddress(ceasar));
88
89     vm.assume(alice != ceasar);

```

```

87
88     vm.assume(token.balanceOf(alice) <= token.totalSupply());
89
90     uint256 oldBalanceAlice = token.balanceOf(alice);
91     uint256 oldBalanceCeasar = token.balanceOf(ceasar);
92     uint256 oldTotalSupply = token.totalSupply();
93
94     vm.prank(alice);
95     token.transfer(alice, amount);
96
97     assertEq(token.balanceOf(alice), oldBalanceAlice);
98     assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
99
100    assertEq(oldTotalSupply, token.totalSupply());
101
102    assertLe(token.balanceOf(alice), token.totalSupply());
103    }
104
105    function testTransfer(address alice, address bob, address ceasar, uint256 amount)
106        public {
107        kevm.infiniteGas();
108
109        vm.assume(alice != address(0));
110        vm.assume(bob != address(0));
111        vm.assume(ceasar != address(0));
112
113        vm.assume(notBuiltinAddress(alice));
114        vm.assume(notBuiltinAddress(bob));
115        vm.assume(notBuiltinAddress(ceasar));
116
117        vm.assume(alice != bob);
118        vm.assume(alice != ceasar);
119        vm.assume(bob != ceasar);
120
121        vm.assume((amount >= token.balanceOf(bob)));
122        vm.assume((amount + token.balanceOf(bob)) <= (token.balanceOf(alice) + token.
123            balanceOf(bob)));
124        vm.assume((token.balanceOf(alice) + token.balanceOf(bob)) <= token.totalSupply());
125
126        uint256 oldBalanceAlice = token.balanceOf(alice);
127        uint256 oldBalanceBob = token.balanceOf(bob);
128        uint256 oldBalanceCeasar = token.balanceOf(ceasar);
129        uint256 oldTotalSupply = token.totalSupply();
130
131        vm.prank(alice);
132        token.transfer(bob, amount);
133
134        assertEq(token.balanceOf(alice), oldBalanceAlice - amount);
135        assertEq(token.balanceOf(bob), oldBalanceBob + amount);
136        assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
137
138        assertEq(token.balanceOf(alice) + token.balanceOf(bob), oldBalanceAlice +
139            oldBalanceBob);
140        assertEq(oldTotalSupply, token.totalSupply());
141
142        assertLe((token.balanceOf(alice) + token.balanceOf(bob)), token.totalSupply());
143    }
144
145    function testEqualTransferFrom(address alice, address ceasar, uint256 amount)
146        public {
147        kevm.infiniteGas();
    
```

```

145
146 vm.assume(alice != address(0));
147 vm.assume(ceasar != address(0));
148
149 vm.assume(notBuiltinAddress(alice));
150 vm.assume(notBuiltinAddress(ceasar));
151
152 vm.assume(alice != ceasar);
153
154 vm.assume((amount >= token.allowance(alice, alice)));
155 vm.assume(token.balanceOf(alice) <= token.totalSupply());
156
157 uint256 oldBalanceAlice = token.balanceOf(alice);
158 uint256 oldBalanceCeasar = token.balanceOf(ceasar);
159 uint256 oldTotalSupply = token.totalSupply();
160
161 vm.prank(alice);
162 token.transferFrom(alice, alice, amount);
163
164 assertEquals(token.balanceOf(alice), oldBalanceAlice);
165 assertEquals(token.balanceOf(ceasar), oldBalanceCeasar);
166
167 assertEquals(oldTotalSupply, token.totalSupply());
168
169 assertLe(token.balanceOf(alice), token.totalSupply());
170 }
171
172 function testTransferFrom(address alice, address bob, address ceasar, address david
    , uint256 amount) public {
173     kevm.infiniteGas();
174
175     vm.assume(alice != address(0));
176     vm.assume(bob != address(0));
177     vm.assume(ceasar != address(0));
178     vm.assume(david != address(0));
179
180     vm.assume(notBuiltinAddress(alice));
181     vm.assume(notBuiltinAddress(bob));
182     vm.assume(notBuiltinAddress(ceasar));
183     vm.assume(notBuiltinAddress(david));
184
185     vm.assume(alice != bob);
186     vm.assume(alice != ceasar);
187     vm.assume(bob != ceasar);
188     vm.assume(bob != david);
189     vm.assume(ceasar != david);
190
191     vm.assume((amount >= token.allowance(alice, david)));
192     vm.assume((amount >= token.balanceOf(bob)));
193     vm.assume((amount + token.balanceOf(bob)) <= (token.balanceOf(alice) + token.
        balanceOf(bob)));
194     vm.assume((token.balanceOf(alice) + token.balanceOf(bob)) <= token.totalSupply());
195
196     uint256 oldBalanceAlice = token.balanceOf(alice);
197     uint256 oldBalanceBob = token.balanceOf(bob);
198     uint256 oldBalanceDavid = token.balanceOf(david);
199     uint256 oldBalanceCeasar = token.balanceOf(ceasar);
200     uint256 oldTotalSupply = token.totalSupply();
201
202     vm.prank(david);
203     token.transferFrom(alice, bob, amount);
204
  
```

```

205     assertEq(token.balanceOf(alice), oldBalanceAlice - amount);
206     assertEq(token.balanceOf(bob), oldBalanceBob + amount);
207     if (alice != david) {
208         assertEq(token.balanceOf(david), oldBalanceDavid);
209     }
210     assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
211
212     assertEq(token.balanceOf(alice) + token.balanceOf(bob), oldBalanceAlice +
        oldBalanceBob);
213     assertEq(oldTotalSupply, token.totalSupply());
214
215     assertLe((token.balanceOf(alice) + token.balanceOf(bob)), token.totalSupply());
216     }
217 }

```

A.3.2 BitteluxFullUnchecked.t.sol

```

1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.13;
3
4 import "../src/BitteluxUnchecked.sol";
5 import "forge-std/Test.sol";
6 import "kontrol-cheatcodes/KontrolCheats.sol";
7
8 contract BitteluxUncheckedTest is Test, KontrolCheats {
9     uint256 constant MAX_INT = 2**256 - 1;
10    BitteluxUnchecked token; // Contract under test
11
12    function setUp() public {
13        token = new BitteluxUnchecked();
14    }
15
16    function testEqualReceive(address alice, address ceasar, uint256 amount) public {
17        kevm.infiniteGas();
18
19        vm.assume(alice != address(0));
20        vm.assume(ceasar != address(0));
21        vm.assume(alice == token.fundsWallet());
22        vm.assume(notBuiltinAddress(alice));
23        vm.assume(notBuiltinAddress(ceasar));
24
25        vm.assume(token.balanceOf(alice) <= token.totalSupply());
26
27        uint256 oldBalanceAlice = token.balanceOf(alice);
28        uint256 oldBalanceCeasar = token.balanceOf(ceasar);
29        uint256 oldTotalSupply = token.totalSupply();
30
31        vm.prank(alice);
32        (bool sent, bytes memory data) = payable(token).call{value: amount}("");
33
34        assertEq(token.balanceOf(alice), oldBalanceAlice);
35        assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
36
37        assertEq(oldTotalSupply, token.totalSupply());
38
39        assertLe(token.balanceOf(alice), token.totalSupply());
40    }
41
42    function testReceive(address alice, address ceasar, uint256 amount) public {
43        kevm.infiniteGas();

```

```

44
45 vm.assume(alice != address(0));
46 vm.assume(ceasar != address(0));
47 vm.assume(token.fundsWallet() != address(0));
48 vm.assume(notBuiltinAddress(alice));
49 vm.assume(alice != token.fundsWallet());
50 vm.assume(notBuiltinAddress(token.fundsWallet()));
51
52 uint256 tokens_bought = amount * token.unitsOneEthCanBuy();
53
54 vm.assume((tokens_bought + token.balanceOf(alice) <= token.balanceOf(token.
    fundsWallet()) + token.balanceOf(alice)));
55 vm.assume(token.balanceOf(token.fundsWallet()) + token.balanceOf(alice) <= token.
    totalSupply());
56
57 uint256 oldBalanceAlice = token.balanceOf(alice);
58 uint256 oldBalanceFundsWallet = token.balanceOf(token.fundsWallet());
59 uint256 oldBalanceCeasar = token.balanceOf(ceasar);
60 uint256 oldTotalSupply = token.totalSupply();
61
62 vm.prank(alice);
63 (bool sent, bytes memory data) = payable(token).call{value: amount}("");
64
65 assertEq(token.balanceOf(alice), oldBalanceAlice + tokens_bought);
66 assertEq(token.balanceOf(token.fundsWallet()), oldBalanceFundsWallet - tokens_bought)
    ;
67 assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
68
69 assertEq(token.balanceOf(alice) + token.balanceOf(token.fundsWallet()),
    oldBalanceAlice + oldBalanceFundsWallet);
70 assertEq(oldTotalSupply, token.totalSupply());
71
72 assertLe((token.balanceOf(alice) + token.balanceOf(token.fundsWallet())), token.
    totalSupply());
73 }
74
75
76
77 function testEqualTransfer(address alice, address ceasar, uint256 amount) public {
78     kevm.infiniteGas();
79
80     vm.assume(alice != address(0));
81     vm.assume(ceasar != address(0));
82
83     vm.assume(notBuiltinAddress(alice));
84     vm.assume(notBuiltinAddress(ceasar));
85
86     vm.assume(alice != ceasar);
87
88     vm.assume(token.balanceOf(alice) <= token.totalSupply());
89
90     uint256 oldBalanceAlice = token.balanceOf(alice);
91     uint256 oldBalanceCeasar = token.balanceOf(ceasar);
92     uint256 oldTotalSupply = token.totalSupply();
93
94     vm.prank(alice);
95     token.transfer(alice, amount);
96
97     assertEq(token.balanceOf(alice), oldBalanceAlice);
98     assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
99
100    assertEq(oldTotalSupply, token.totalSupply());
    
```

```

101
102     assertLe(token.balanceOf(alice), token.totalSupply());
103     }
104
105     function testTransfer(address alice, address bob, address ceasar, uint256 amount)
106         public {
107         kevm.infiniteGas();
108
109         vm.assume(alice != address(0));
110         vm.assume(bob != address(0));
111         vm.assume(ceasar != address(0));
112
113         vm.assume(notBuiltinAddress(alice));
114         vm.assume(notBuiltinAddress(bob));
115         vm.assume(notBuiltinAddress(ceasar));
116
117         vm.assume(alice != bob);
118         vm.assume(alice != ceasar);
119         vm.assume(bob != ceasar);
120
121         vm.assume((amount >= token.balanceOf(bob)));
122         vm.assume((amount + token.balanceOf(bob)) <= (token.balanceOf(alice) + token.
123             balanceOf(bob)));
124         vm.assume((token.balanceOf(alice) + token.balanceOf(bob)) <= token.totalSupply());
125
126         uint256 oldBalanceAlice = token.balanceOf(alice);
127         uint256 oldBalanceBob = token.balanceOf(bob);
128         uint256 oldBalanceCeasar = token.balanceOf(ceasar);
129         uint256 oldTotalSupply = token.totalSupply();
130
131         vm.prank(alice);
132         token.transfer(bob, amount);
133
134         assertEq(token.balanceOf(alice), oldBalanceAlice - amount);
135         assertEq(token.balanceOf(bob), oldBalanceBob + amount);
136         assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
137
138         assertEq(token.balanceOf(alice) + token.balanceOf(bob), oldBalanceAlice +
139             oldBalanceBob);
140         assertEq(oldTotalSupply, token.totalSupply());
141
142         assertLe((token.balanceOf(alice) + token.balanceOf(bob)), token.totalSupply());
143     }
144
145     function testEqualTransferFrom(address alice, address ceasar, uint256 amount)
146         public {
147         kevm.infiniteGas();
148
149         vm.assume(alice != address(0));
150         vm.assume(ceasar != address(0));
151
152         vm.assume(notBuiltinAddress(alice));
153         vm.assume(notBuiltinAddress(ceasar));
154
155         vm.assume(alice != ceasar);
156
157         vm.assume((amount >= token.allowance(alice, alice)));
158         vm.assume(token.balanceOf(alice) <= token.totalSupply());
159
160         uint256 oldBalanceAlice = token.balanceOf(alice);
161         uint256 oldBalanceCeasar = token.balanceOf(ceasar);
    
```

```

159     uint256 oldTotalSupply = token.totalSupply();
160
161     vm.prank(alice);
162     token.transferFrom(alice, alice, amount);
163
164     assertEquals(token.balanceOf(alice), oldBalanceAlice);
165     assertEquals(token.balanceOf(ceasar), oldBalanceCeasar);
166
167     assertEquals(oldTotalSupply, token.totalSupply());
168
169     assertLe(token.balanceOf(alice), token.totalSupply());
170 }
171
172     function testTransferFrom(address alice, address bob, address ceasar, address david
173         , uint256 amount) public {
174         kevm.infiniteGas();
175
176         vm.assume(alice != address(0));
177         vm.assume(bob != address(0));
178         vm.assume(ceasar != address(0));
179         vm.assume(david != address(0));
180
181         vm.assume(notBuiltinAddress(alice));
182         vm.assume(notBuiltinAddress(bob));
183         vm.assume(notBuiltinAddress(ceasar));
184         vm.assume(notBuiltinAddress(david));
185
186         vm.assume(alice != bob);
187         vm.assume(alice != ceasar);
188         vm.assume(bob != ceasar);
189         vm.assume(bob != david);
190         vm.assume(ceasar != david);
191
192         vm.assume((amount >= token.allowance(alice, david)));
193         vm.assume((amount >= token.balanceOf(bob)));
194         vm.assume((amount + token.balanceOf(bob)) <= (token.balanceOf(alice) + token.
195             balanceOf(bob)));
196         vm.assume((token.balanceOf(alice) + token.balanceOf(bob)) <= token.totalSupply());
197
198         uint256 oldBalanceAlice = token.balanceOf(alice);
199         uint256 oldBalanceBob = token.balanceOf(bob);
200         uint256 oldBalanceDavid = token.balanceOf(david);
201         uint256 oldBalanceCeasar = token.balanceOf(ceasar);
202         uint256 oldTotalSupply = token.totalSupply();
203
204         vm.prank(david);
205         token.transferFrom(alice, bob, amount);
206
207         assertEquals(token.balanceOf(alice), oldBalanceAlice - amount);
208         assertEquals(token.balanceOf(bob), oldBalanceBob + amount);
209         assertEquals(token.balanceOf(ceasar), oldBalanceCeasar);
210         if (alice != david) {
211             assertEquals(token.balanceOf(david), oldBalanceDavid);
212         }
213
214         assertEquals(token.balanceOf(alice) + token.balanceOf(bob), oldBalanceAlice +
215             oldBalanceBob);
216         assertEquals(oldTotalSupply, token.totalSupply());
217
218         assertLe((token.balanceOf(alice) + token.balanceOf(bob)), token.totalSupply());
219     }
220 }

```

A.3.3 CCLAGFull.t.sol

```
1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.13;
3
4 import "../src/CCLAG.sol";
5 import "forge-std/Test.sol";
6 import "kontrol-cheatcodes/KontrolCheats.sol";
7
8 contract ChuCunLingAIGOTest is Test, KontrolCheats {
9     uint256 constant MAX_INT = 2**256 - 1;
10    ChuCunLingAIGO token; // Contract under test
11
12    function setUp() public {
13        token = new ChuCunLingAIGO(100000000000000000, "ChuCunLingAIGO", 4, "CCLAG");
14    }
15
16    function testEqualTransfer(address alice, address ceasar, uint256 amount) public {
17        kevm.infiniteGas();
18
19        vm.assume(alice != address(0));
20        vm.assume(ceasar != address(0));
21
22        vm.assume(notBuiltinAddress(alice));
23        vm.assume(notBuiltinAddress(ceasar));
24
25        vm.assume(alice != ceasar);
26
27        vm.assume(token.balanceOf(alice) <= token.totalSupply());
28
29        uint256 oldBalanceAlice = token.balanceOf(alice);
30        uint256 oldBalanceCeasar = token.balanceOf(ceasar);
31        uint256 oldTotalSupply = token.totalSupply();
32
33        vm.prank(alice);
34        token.transfer(alice, amount);
35
36        assertEq(token.balanceOf(alice), oldBalanceAlice);
37        assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
38
39        assertEq(oldTotalSupply, token.totalSupply());
40
41        assertLe(token.balanceOf(alice), token.totalSupply());
42    }
43
44    function testTransfer(address alice, address bob, address ceasar, uint256 amount)
45        public {
46        kevm.infiniteGas();
47
48        vm.assume(alice != address(0));
49        vm.assume(bob != address(0));
50        vm.assume(ceasar != address(0));
51
52        vm.assume(notBuiltinAddress(alice));
53        vm.assume(notBuiltinAddress(bob));
54        vm.assume(notBuiltinAddress(ceasar));
55
56        vm.assume(alice != bob);
57        vm.assume(alice != ceasar);
58        vm.assume(bob != ceasar);
59
60        vm.assume((amount >= token.balanceOf(bob)));
```

```

60     vm.assume((amount + token.balanceOf(bob)) <= (token.balanceOf(alice) + token.
        balanceOf(bob)));
61     vm.assume((token.balanceOf(alice) + token.balanceOf(bob)) <= token.totalSupply());
62
63     uint256 oldBalanceAlice = token.balanceOf(alice);
64     uint256 oldBalanceBob = token.balanceOf(bob);
65     uint256 oldBalanceCeasar = token.balanceOf(ceasar);
66     uint256 oldTotalSupply = token.totalSupply();
67
68     vm.prank(alice);
69     token.transfer(bob, amount);
70
71     assertEq(token.balanceOf(alice), oldBalanceAlice - amount);
72     assertEq(token.balanceOf(bob), oldBalanceBob + amount);
73     assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
74
75     assertEq(token.balanceOf(alice) + token.balanceOf(bob), oldBalanceAlice +
        oldBalanceBob);
76     assertEq(oldTotalSupply, token.totalSupply());
77
78     assertLe((token.balanceOf(alice) + token.balanceOf(bob)), token.totalSupply());
79     }
80
81
82     function testEqualTransferFrom(address alice, address ceasar, uint256 amount)
        public {
83         kevm.infiniteGas();
84
85         vm.assume(alice != address(0));
86         vm.assume(ceasar != address(0));
87
88         vm.assume(notBuiltinAddress(alice));
89         vm.assume(notBuiltinAddress(ceasar));
90
91         vm.assume(alice != ceasar);
92
93         vm.assume((amount >= token.allowance(alice, alice)));
94         vm.assume(token.balanceOf(alice) <= token.totalSupply());
95
96         uint256 oldBalanceAlice = token.balanceOf(alice);
97         uint256 oldBalanceCeasar = token.balanceOf(ceasar);
98         uint256 oldTotalSupply = token.totalSupply();
99
100        vm.prank(alice);
101        token.transferFrom(alice, ceasar, amount);
102
103        assertEq(token.balanceOf(alice), oldBalanceAlice);
104        assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
105
106        assertEq(oldTotalSupply, token.totalSupply());
107
108        assertLe(token.balanceOf(alice), token.totalSupply());
109    }
110
111    function testTransferFrom(address alice, address bob, address ceasar, address david
        , uint256 amount) public {
112        kevm.infiniteGas();
113
114        vm.assume(alice != address(0));
115        vm.assume(bob != address(0));
116        vm.assume(ceasar != address(0));
117        vm.assume(david != address(0));
    
```

```

118
119     vm.assume(notBuiltinAddress(alice));
120     vm.assume(notBuiltinAddress(bob));
121     vm.assume(notBuiltinAddress(ceasar));
122     vm.assume(notBuiltinAddress(david));
123
124     vm.assume(alice != bob);
125     vm.assume(alice != ceasar);
126     vm.assume(bob != ceasar);
127     vm.assume(bob != david);
128     vm.assume(ceasar != david);
129
130     vm.assume((amount >= token.allowance(alice, david)));
131     vm.assume((amount >= token.balanceOf(bob)));
132     vm.assume((amount + token.balanceOf(bob)) <= (token.balanceOf(alice) + token.
        balanceOf(bob)));
133     vm.assume((token.balanceOf(alice) + token.balanceOf(bob)) <= token.totalSupply());
134
135     uint256 oldBalanceAlice = token.balanceOf(alice);
136     uint256 oldBalanceBob = token.balanceOf(bob);
137     uint256 oldBalanceDavid = token.balanceOf(david);
138     uint256 oldBalanceCeasar = token.balanceOf(ceasar);
139     uint256 oldTotalSupply = token.totalSupply();
140
141     vm.prank(david);
142     token.transferFrom(alice, bob, amount);
143
144     assertEq(token.balanceOf(alice), oldBalanceAlice - amount);
145     assertEq(token.balanceOf(bob), oldBalanceBob + amount);
146     if (alice != david) {
147         assertEq(token.balanceOf(david), oldBalanceDavid);
148     }
149     assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
150
151     assertEq(token.balanceOf(alice) + token.balanceOf(bob), oldBalanceAlice +
        oldBalanceBob);
152     assertEq(oldTotalSupply, token.totalSupply());
153
154     assertLe((token.balanceOf(alice) + token.balanceOf(bob)), token.totalSupply());
155 }
156 }

```

A.3.4 CCLAGFullUnchecked.t.sol

```

1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.13;
3
4 import "../src/CCLAGUnchecked.sol";
5 import "forge-std/Test.sol";
6 import "kontrol-cheatcodes/KontrolCheats.sol";
7
8 contract ChuCunLingAIGOUNcheckedTest is Test, KontrolCheats {
9     uint256 constant MAX_INT = 2**256 - 1;
10     ChuCunLingAIGOUNchecked token; // Contract under test
11
12     function setUp() public {
13         token = new ChuCunLingAIGOUNchecked(10000000000000000, "ChuCunLingAIGO", 4, "
            CCLAG");
14     }
15 }

```

```

16     function testEqualTransfer(address alice, address ceasar, uint256 amount) public {
17         kevm.infiniteGas();
18
19         vm.assume(alice != address(0));
20         vm.assume(ceasar != address(0));
21
22         vm.assume(notBuiltinAddress(alice));
23         vm.assume(notBuiltinAddress(ceasar));
24
25         vm.assume(alice != ceasar);
26
27         vm.assume(token.balanceOf(alice) <= token.totalSupply());
28
29         uint256 oldBalanceAlice = token.balanceOf(alice);
30         uint256 oldBalanceCeasar = token.balanceOf(ceasar);
31         uint256 oldTotalSupply = token.totalSupply();
32
33         vm.prank(alice);
34         token.transfer(alice, amount);
35
36         assertEquals(token.balanceOf(alice), oldBalanceAlice);
37         assertEquals(token.balanceOf(ceasar), oldBalanceCeasar);
38
39         assertEquals(oldTotalSupply, token.totalSupply());
40
41         assertLe(token.balanceOf(alice), token.totalSupply());
42     }
43
44     function testTransfer(address alice, address bob, address ceasar, uint256 amount)
45         public {
46         kevm.infiniteGas();
47
48         vm.assume(alice != address(0));
49         vm.assume(bob != address(0));
50         vm.assume(ceasar != address(0));
51
52         vm.assume(notBuiltinAddress(alice));
53         vm.assume(notBuiltinAddress(bob));
54         vm.assume(notBuiltinAddress(ceasar));
55
56         vm.assume(alice != bob);
57         vm.assume(alice != ceasar);
58         vm.assume(bob != ceasar);
59
60         vm.assume((amount >= token.balanceOf(bob)));
61         vm.assume((amount + token.balanceOf(bob) <= (token.balanceOf(alice) + token.
62             balanceOf(bob))));
63         vm.assume((token.balanceOf(alice) + token.balanceOf(bob)) <= token.totalSupply());
64
65         uint256 oldBalanceAlice = token.balanceOf(alice);
66         uint256 oldBalanceBob = token.balanceOf(bob);
67         uint256 oldBalanceCeasar = token.balanceOf(ceasar);
68         uint256 oldTotalSupply = token.totalSupply();
69
70         vm.prank(alice);
71         token.transfer(bob, amount);
72
73         assertEquals(token.balanceOf(alice), oldBalanceAlice - amount);
74         assertEquals(token.balanceOf(bob), oldBalanceBob + amount);
75         assertEquals(token.balanceOf(ceasar), oldBalanceCeasar);
76
77         assertEquals(token.balanceOf(alice) + token.balanceOf(bob), oldBalanceAlice +
    
```

```

76     oldBalanceBob);
77     assertEq(oldTotalSupply, token.totalSupply());
78
79     assertLe((token.balanceOf(alice) + token.balanceOf(bob)), token.totalSupply());
80 }
81
82     function testEqualTransferFrom(address alice, address ceasar, uint256 amount)
83     public {
84         kevm.infiniteGas();
85
86         vm.assume(alice != address(0));
87         vm.assume(ceasar != address(0));
88
89         vm.assume(notBuiltinAddress(alice));
90         vm.assume(notBuiltinAddress(ceasar));
91
92         vm.assume(alice != ceasar);
93
94         vm.assume((amount >= token.allowance(alice, alice)));
95         vm.assume(token.balanceOf(alice) <= token.totalSupply());
96
97         uint256 oldBalanceAlice = token.balanceOf(alice);
98         uint256 oldBalanceCeasar = token.balanceOf(ceasar);
99         uint256 oldTotalSupply = token.totalSupply();
100
101         vm.prank(alice);
102         token.transferFrom(alice, alice, amount);
103
104         assertEq(token.balanceOf(alice), oldBalanceAlice);
105         assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
106
107         assertEq(oldTotalSupply, token.totalSupply());
108
109         assertLe(token.balanceOf(alice), token.totalSupply());
110     }
111
112     function testTransferFrom(address alice, address bob, address ceasar, address david
113     , uint256 amount) public {
114         kevm.infiniteGas();
115
116         vm.assume(alice != address(0));
117         vm.assume(bob != address(0));
118         vm.assume(ceasar != address(0));
119         vm.assume(david != address(0));
120
121         vm.assume(notBuiltinAddress(alice));
122         vm.assume(notBuiltinAddress(bob));
123         vm.assume(notBuiltinAddress(ceasar));
124         vm.assume(notBuiltinAddress(david));
125
126         vm.assume(alice != bob);
127         vm.assume(alice != ceasar);
128         vm.assume(bob != ceasar);
129         vm.assume(bob != david);
130         vm.assume(ceasar != david);
131
132         vm.assume((amount >= token.allowance(alice, david)));
133         vm.assume((amount >= token.balanceOf(bob)));
134         vm.assume((amount + token.balanceOf(bob)) <= (token.balanceOf(alice) + token.
135             balanceOf(bob)));
136         vm.assume((token.balanceOf(alice) + token.balanceOf(bob)) <= token.totalSupply());

```

```

134
135     uint256 oldBalanceAlice = token.balanceOf(alice);
136     uint256 oldBalanceBob = token.balanceOf(bob);
137     uint256 oldBalanceDavid = token.balanceOf(david);
138     uint256 oldBalanceCeasar = token.balanceOf(ceasar);
139     uint256 oldTotalSupply = token.totalSupply();
140
141     vm.prank(david);
142     token.transferFrom(alice, bob, amount);
143
144     assertEq(token.balanceOf(alice), oldBalanceAlice - amount);
145     assertEq(token.balanceOf(bob), oldBalanceBob + amount);
146     assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
147     if (alice != david) {
148         assertEq(token.balanceOf(david), oldBalanceDavid);
149     }
150
151     assertEq(token.balanceOf(alice) + token.balanceOf(bob), oldBalanceAlice +
152         oldBalanceBob);
153     assertEq(oldTotalSupply, token.totalSupply());
154
155     assertLe((token.balanceOf(alice) + token.balanceOf(bob)), token.totalSupply());
156 }
  
```

A.3.5 ETTFull.t.sol

```

1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.13;
3
4 import "../src/ETT.sol";
5 import "forge-std/Test.sol";
6 import "kontrol-cheatcodes/KontrolCheats.sol";
7
8 contract HashnodeTestCoinTest is Test, KontrolCheats {
9     uint256 constant MAX_INT = 2**256 - 1;
10    HashnodeTestCoin token; // Contract under test
11
12    function setUp() public {
13        token = new HashnodeTestCoin();
14    }
15
16    function testEqualReceive(address alice, address ceasar, uint256 amount) public {
17        kevm.infiniteGas();
18
19        vm.assume(alice != address(0));
20        vm.assume(ceasar != address(0));
21        vm.assume(alice == token.fundsWallet());
22        vm.assume(notBuiltinAddress(alice));
23        vm.assume(notBuiltinAddress(ceasar));
24
25        vm.assume(token.balanceOf(alice) <= token.totalSupply());
26
27        uint256 oldBalanceAlice = token.balanceOf(alice);
28        uint256 oldBalanceCeasar = token.balanceOf(ceasar);
29        uint256 oldTotalSupply = token.totalSupply();
30
31        vm.prank(alice);
32        (bool sent, bytes memory data) = payable(token).call{value: amount}("");
33
  
```

```

34  assertEq(token.balanceOf(alice), oldBalanceAlice);
35  assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
36
37  assertEq(oldTotalSupply, token.totalSupply());
38
39  assertLe(token.balanceOf(alice), token.totalSupply());
40  }
41
42  function testReceive(address alice, address ceasar, uint256 amount) public {
43      kevm.infiniteGas();
44
45      vm.assume(alice != address(0));
46      vm.assume(ceasar != address(0));
47      vm.assume(token.fundsWallet() != address(0));
48      vm.assume(notBuiltinAddress(alice));
49      vm.assume(alice != token.fundsWallet());
50      vm.assume(notBuiltinAddress(token.fundsWallet()));
51
52      uint256 tokens_bought = amount * token.unitsOneEthCanBuy();
53
54      vm.assume((tokens_bought + token.balanceOf(alice) <= token.balanceOf(token.
55          fundsWallet()) + token.balanceOf(alice)));
56      vm.assume(token.balanceOf(token.fundsWallet()) + token.balanceOf(alice) <= token.
57          totalSupply());
58
59      uint256 oldBalanceAlice = token.balanceOf(alice);
60      uint256 oldBalanceFundsWallet = token.balanceOf(token.fundsWallet());
61      uint256 oldBalanceCeasar = token.balanceOf(ceasar);
62      uint256 oldTotalSupply = token.totalSupply();
63
64      vm.prank(alice);
65      (bool sent, bytes memory data) = payable(token).call{value: amount}("");
66
67      assertEq(token.balanceOf(alice), oldBalanceAlice + tokens_bought);
68      assertEq(token.balanceOf(token.fundsWallet()), oldBalanceFundsWallet - tokens_bought)
69      ;
70      assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
71
72      assertEq(token.balanceOf(alice) + token.balanceOf(token.fundsWallet()),
73          oldBalanceAlice + oldBalanceFundsWallet);
74      assertEq(oldTotalSupply, token.totalSupply());
75
76      assertLe((token.balanceOf(alice) + token.balanceOf(token.fundsWallet())), token.
77          totalSupply());
78  }
79
80  function testEqualTransfer(address alice, address ceasar, uint256 amount) public {
81      kevm.infiniteGas();
82
83      vm.assume(alice != address(0));
84      vm.assume(ceasar != address(0));
85
86      vm.assume(notBuiltinAddress(alice));
87      vm.assume(notBuiltinAddress(ceasar));
88
89      vm.assume(alice != ceasar);
90
91      vm.assume(token.balanceOf(alice) <= token.totalSupply());
92
93      uint256 oldBalanceAlice = token.balanceOf(alice);
    
```

```

91     uint256 oldBalanceCeasar = token.balanceOf(ceasar);
92     uint256 oldTotalSupply = token.totalSupply();
93
94     vm.prank(alice);
95     token.transfer(alice, amount);
96
97     assertEquals(token.balanceOf(alice), oldBalanceAlice);
98     assertEquals(token.balanceOf(ceasar), oldBalanceCeasar);
99
100    assertEquals(oldTotalSupply, token.totalSupply());
101
102    assertLe(token.balanceOf(alice), token.totalSupply());
103    }
104
105    function testTransfer(address alice, address bob, address ceasar, uint256 amount)
106        public {
107        kevm.infiniteGas();
108
109        vm.assume(alice != address(0));
110        vm.assume(bob != address(0));
111        vm.assume(ceasar != address(0));
112
113        vm.assume(notBuiltinAddress(alice));
114        vm.assume(notBuiltinAddress(bob));
115        vm.assume(notBuiltinAddress(ceasar));
116
117        vm.assume(alice != bob);
118        vm.assume(alice != ceasar);
119        vm.assume(bob != ceasar);
120
121        vm.assume((amount >= token.balanceOf(bob)));
122        vm.assume((amount + token.balanceOf(bob)) <= (token.balanceOf(alice) + token.
123            balanceOf(bob)));
124        vm.assume((token.balanceOf(alice) + token.balanceOf(bob)) <= token.totalSupply());
125
126        uint256 oldBalanceAlice = token.balanceOf(alice);
127        uint256 oldBalanceBob = token.balanceOf(bob);
128        uint256 oldBalanceCeasar = token.balanceOf(ceasar);
129        uint256 oldTotalSupply = token.totalSupply();
130
131        vm.prank(alice);
132        token.transfer(bob, amount);
133
134        assertEquals(token.balanceOf(alice), oldBalanceAlice - amount);
135        assertEquals(token.balanceOf(bob), oldBalanceBob + amount);
136        assertEquals(token.balanceOf(ceasar), oldBalanceCeasar);
137
138        assertEquals(token.balanceOf(alice) + token.balanceOf(bob), oldBalanceAlice +
139            oldBalanceBob);
140        assertEquals(oldTotalSupply, token.totalSupply());
141
142        assertLe((token.balanceOf(alice) + token.balanceOf(bob)), token.totalSupply());
143    }
144
145    function testEqualTransferFrom(address alice, address ceasar, uint256 amount)
146        public {
147        kevm.infiniteGas();
148
149        vm.assume(alice != address(0));
150        vm.assume(ceasar != address(0));

```

```

149 vm.assume(notBuiltinAddress(alice));
150 vm.assume(notBuiltinAddress(ceasar));
151
152 vm.assume(alice != ceasar);
153
154 vm.assume((amount >= token.allowance(alice, alice)));
155 vm.assume(token.balanceOf(alice) <= token.totalSupply());
156
157 uint256 oldBalanceAlice = token.balanceOf(alice);
158 uint256 oldBalanceCeasar = token.balanceOf(ceasar);
159 uint256 oldTotalSupply = token.totalSupply();
160
161 vm.prank(alice);
162 token.transferFrom(alice, alice, amount);
163
164 assertEq(token.balanceOf(alice), oldBalanceAlice);
165 assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
166
167 assertEq(oldTotalSupply, token.totalSupply());
168
169 assertLe(token.balanceOf(alice), token.totalSupply());
170 }
171
172 function testTransferFrom(address alice, address bob, address ceasar, address david
    , uint256 amount) public {
173     kevm.infiniteGas();
174
175     vm.assume(alice != address(0));
176     vm.assume(bob != address(0));
177     vm.assume(ceasar != address(0));
178     vm.assume(david != address(0));
179
180     vm.assume(notBuiltinAddress(alice));
181     vm.assume(notBuiltinAddress(bob));
182     vm.assume(notBuiltinAddress(ceasar));
183     vm.assume(notBuiltinAddress(david));
184
185     vm.assume(alice != bob);
186     vm.assume(alice != ceasar);
187     vm.assume(bob != ceasar);
188     vm.assume(bob != david);
189     vm.assume(ceasar != david);
190
191     vm.assume((amount >= token.allowance(alice, david)));
192     vm.assume((amount >= token.balanceOf(bob)));
193     vm.assume((amount + token.balanceOf(bob) <= (token.balanceOf(alice) + token.
        balanceOf(bob)));
194     vm.assume((token.balanceOf(alice) + token.balanceOf(bob)) <= token.totalSupply());
195
196     uint256 oldBalanceAlice = token.balanceOf(alice);
197     uint256 oldBalanceBob = token.balanceOf(bob);
198     uint256 oldBalanceDavid = token.balanceOf(david);
199     uint256 oldBalanceCeasar = token.balanceOf(ceasar);
200     uint256 oldTotalSupply = token.totalSupply();
201
202     vm.prank(david);
203     token.transferFrom(alice, bob, amount);
204
205     assertEq(token.balanceOf(alice), oldBalanceAlice - amount);
206     assertEq(token.balanceOf(bob), oldBalanceBob + amount);
207     if (alice != david) {
208         assertEq(token.balanceOf(david), oldBalanceDavid);
  
```

```

209     }
210     assertEquals(token.balanceOf(ceasar), oldBalanceCeasar);
211
212     assertEquals(token.balanceOf(alice) + token.balanceOf(bob), oldBalanceAlice +
        oldBalanceBob);
213     assertEquals(oldTotalSupply, token.totalSupply());
214
215     assertLe((token.balanceOf(alice) + token.balanceOf(bob)), token.totalSupply());
216     }
217 }

```

A.3.6 ETTFullUnchecked.t.sol

```

1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.13;
3
4 import "../src/ETTUnchecked.sol";
5 import "forge-std/Test.sol";
6 import "kontrol-cheatcodes/KontrolCheats.sol";
7
8 contract HashnodeTestCoinUncheckedTest is Test, KontrolCheats {
9     uint256 constant MAX_INT = 2**256 - 1;
10    HashnodeTestCoinUnchecked token; // Contract under test
11
12    function setUp() public {
13        token = new HashnodeTestCoinUnchecked();
14    }
15
16    function testEqualReceive(address alice, address ceasar, uint256 amount) public {
17        kevm.infiniteGas();
18
19        vm.assume(alice != address(0));
20        vm.assume(ceasar != address(0));
21        vm.assume(alice == token.fundsWallet());
22        vm.assume(notBuiltinAddress(alice));
23        vm.assume(notBuiltinAddress(ceasar));
24
25        vm.assume(token.balanceOf(alice) <= token.totalSupply());
26
27        uint256 oldBalanceAlice = token.balanceOf(alice);
28        uint256 oldBalanceCeasar = token.balanceOf(ceasar);
29        uint256 oldTotalSupply = token.totalSupply();
30
31        vm.prank(alice);
32        (bool sent, bytes memory data) = payable(token).call{value: amount}("");
33
34        assertEquals(token.balanceOf(alice), oldBalanceAlice);
35        assertEquals(token.balanceOf(ceasar), oldBalanceCeasar);
36
37        assertEquals(oldTotalSupply, token.totalSupply());
38
39        assertLe(token.balanceOf(alice), token.totalSupply());
40    }
41
42    function testReceive(address alice, address ceasar, uint256 amount) public {
43        kevm.infiniteGas();
44
45        vm.assume(alice != address(0));
46        vm.assume(ceasar != address(0));
47        vm.assume(token.fundsWallet() != address(0));

```

```

48 vm.assume(notBuiltinAddress(alice));
49 vm.assume(alice != token.fundsWallet());
50 vm.assume(notBuiltinAddress(token.fundsWallet()));
51
52 uint256 tokens_bought = amount * token.unitsOneEthCanBuy();
53
54 vm.assume((tokens_bought + token.balanceOf(alice) <= token.balanceOf(token.
    fundsWallet()) + token.balanceOf(alice)));
55 vm.assume(token.balanceOf(token.fundsWallet()) + token.balanceOf(alice) <= token.
    totalSupply());
56
57 uint256 oldBalanceAlice = token.balanceOf(alice);
58 uint256 oldBalanceFundsWallet = token.balanceOf(token.fundsWallet());
59 uint256 oldBalanceCeasar = token.balanceOf(ceasar);
60 uint256 oldTotalSupply = token.totalSupply();
61
62 vm.prank(alice);
63 (bool sent, bytes memory data) = payable(token).call{value: amount}("");
64
65 assertEq(token.balanceOf(alice), oldBalanceAlice + tokens_bought);
66 assertEq(token.balanceOf(token.fundsWallet()), oldBalanceFundsWallet - tokens_bought)
    ;
67 assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
68
69 assertEq(token.balanceOf(alice) + token.balanceOf(token.fundsWallet()),
    oldBalanceAlice + oldBalanceFundsWallet);
70 assertEq(oldTotalSupply, token.totalSupply());
71
72 assertLe((token.balanceOf(alice) + token.balanceOf(token.fundsWallet())), token.
    totalSupply());
73 }
74
75
76
77 function testEqualTransfer(address alice, address ceasar, uint256 amount) public {
78     kevm.infiniteGas();
79
80     vm.assume(alice != address(0));
81     vm.assume(ceasar != address(0));
82
83     vm.assume(notBuiltinAddress(alice));
84     vm.assume(notBuiltinAddress(ceasar));
85
86     vm.assume(alice != ceasar);
87
88     vm.assume(token.balanceOf(alice) <= token.totalSupply());
89
90     uint256 oldBalanceAlice = token.balanceOf(alice);
91     uint256 oldBalanceCeasar = token.balanceOf(ceasar);
92     uint256 oldTotalSupply = token.totalSupply();
93
94     vm.prank(alice);
95     token.transfer(alice, amount);
96
97     assertEq(token.balanceOf(alice), oldBalanceAlice);
98     assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
99
100     assertEq(oldTotalSupply, token.totalSupply());
101
102     assertLe(token.balanceOf(alice), token.totalSupply());
103 }
104

```

```

105     function testTransfer(address alice, address bob, address ceasar, uint256 amount)
106         public {
107             kevm.infiniteGas();
108             vm.assume(alice != address(0));
109             vm.assume(bob != address(0));
110             vm.assume(ceasar != address(0));
111
112             vm.assume(notBuiltinAddress(alice));
113             vm.assume(notBuiltinAddress(bob));
114             vm.assume(notBuiltinAddress(ceasar));
115
116             vm.assume(alice != bob);
117             vm.assume(alice != ceasar);
118             vm.assume(bob != ceasar);
119
120             vm.assume((amount >= token.balanceOf(bob)));
121             vm.assume((amount + token.balanceOf(bob)) <= (token.balanceOf(alice) + token.
122                 balanceOf(bob)));
123             vm.assume((token.balanceOf(alice) + token.balanceOf(bob)) <= token.totalSupply());
124             uint256 oldBalanceAlice = token.balanceOf(alice);
125             uint256 oldBalanceBob = token.balanceOf(bob);
126             uint256 oldBalanceCeasar = token.balanceOf(ceasar);
127             uint256 oldTotalSupply = token.totalSupply();
128
129             vm.prank(alice);
130             token.transfer(bob, amount);
131
132             assertEq(token.balanceOf(alice), oldBalanceAlice - amount);
133             assertEq(token.balanceOf(bob), oldBalanceBob + amount);
134             assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
135
136             assertEq(token.balanceOf(alice) + token.balanceOf(bob), oldBalanceAlice +
137                 oldBalanceBob);
138             assertEq(oldTotalSupply, token.totalSupply());
139             assertLe((token.balanceOf(alice) + token.balanceOf(bob)), token.totalSupply());
140         }
141
142
143     function testEqualTransferFrom(address alice, address ceasar, uint256 amount)
144         public {
145             kevm.infiniteGas();
146             vm.assume(alice != address(0));
147             vm.assume(ceasar != address(0));
148
149             vm.assume(notBuiltinAddress(alice));
150             vm.assume(notBuiltinAddress(ceasar));
151
152             vm.assume(alice != ceasar);
153
154             vm.assume((amount >= token.allowance(alice, alice)));
155             vm.assume(token.balanceOf(alice) <= token.totalSupply());
156
157             uint256 oldBalanceAlice = token.balanceOf(alice);
158             uint256 oldBalanceCeasar = token.balanceOf(ceasar);
159             uint256 oldTotalSupply = token.totalSupply();
160
161             vm.prank(alice);
162             token.transferFrom(alice, ceasar, amount);

```

```

163
164     assertEquals(token.balanceOf(alice), oldBalanceAlice);
165     assertEquals(token.balanceOf(ceasar), oldBalanceCeasar);
166
167     assertEquals(oldTotalSupply, token.totalSupply());
168
169     assertLe(token.balanceOf(alice), token.totalSupply());
170 }
171
172     function testTransferFrom(address alice, address bob, address ceasar, address david
173         , uint256 amount) public {
174         kevm.infiniteGas();
175
176         vm.assume(alice != address(0));
177         vm.assume(bob != address(0));
178         vm.assume(ceasar != address(0));
179         vm.assume(david != address(0));
180
181         vm.assume(notBuiltinAddress(alice));
182         vm.assume(notBuiltinAddress(bob));
183         vm.assume(notBuiltinAddress(ceasar));
184         vm.assume(notBuiltinAddress(david));
185
186         vm.assume(alice != bob);
187         vm.assume(alice != ceasar);
188         vm.assume(bob != ceasar);
189         vm.assume(bob != david);
190         vm.assume(ceasar != david);
191
192         vm.assume((amount >= token.allowance(alice, david)));
193         vm.assume((amount >= token.balanceOf(bob)));
194         vm.assume((amount + token.balanceOf(bob)) <= (token.balanceOf(alice) + token.
195             balanceOf(bob)));
196         vm.assume((token.balanceOf(alice) + token.balanceOf(bob)) <= token.totalSupply());
197
198         uint256 oldBalanceAlice = token.balanceOf(alice);
199         uint256 oldBalanceBob = token.balanceOf(bob);
200         uint256 oldBalanceDavid = token.balanceOf(david);
201         uint256 oldBalanceCeasar = token.balanceOf(ceasar);
202         uint256 oldTotalSupply = token.totalSupply();
203
204         vm.prank(david);
205         token.transferFrom(alice, bob, amount);
206
207         assertEquals(token.balanceOf(alice), oldBalanceAlice - amount);
208         assertEquals(token.balanceOf(bob), oldBalanceBob + amount);
209         assertEquals(token.balanceOf(ceasar), oldBalanceCeasar);
210         if (alice != david) {
211             assertEquals(token.balanceOf(david), oldBalanceDavid);
212         }
213
214         assertEquals(token.balanceOf(alice) + token.balanceOf(bob), oldBalanceAlice +
215             oldBalanceBob);
216         assertEquals(oldTotalSupply, token.totalSupply());
217
218         assertLe((token.balanceOf(alice) + token.balanceOf(bob)), token.totalSupply());
219     }

```

A.3.7 PandoraFull.t.sol

```

1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.13;
3
4 import "../src/Pandora.sol";
5 import "forge-std/Test.sol";
6 import "kontrol-cheatcodes/KontrolCheats.sol";
7
8 contract HumanStandardTokenTest is Test, KontrolCheats {
9     uint256 constant MAX_INT = 2**256 - 1;
10    HumanStandardToken token; // Contract under test
11
12    function setUp() public {
13        token = new HumanStandardToken(1000000000000000, "Pandora", 4, "PD");
14    }
15
16    function testEqualTransfer(address alice, address ceasar, uint256 amount) public {
17        kevm.infiniteGas();
18
19        vm.assume(alice != address(0));
20        vm.assume(ceasar != address(0));
21
22        vm.assume(notBuiltinAddress(alice));
23        vm.assume(notBuiltinAddress(ceasar));
24
25        vm.assume(alice != ceasar);
26
27        vm.assume(token.balanceOf(alice) <= token.totalSupply());
28
29        uint256 oldBalanceAlice = token.balanceOf(alice);
30        uint256 oldBalanceCeasar = token.balanceOf(ceasar);
31        uint256 oldTotalSupply = token.totalSupply();
32
33        vm.prank(alice);
34        token.transfer(alice, amount);
35
36        assertEq(token.balanceOf(alice), oldBalanceAlice);
37        assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
38
39        assertEq(oldTotalSupply, token.totalSupply());
40
41        assertLe(token.balanceOf(alice), token.totalSupply());
42    }
43
44    function testTransfer(address alice, address bob, address ceasar, uint256 amount)
45        public {
46        kevm.infiniteGas();
47
48        vm.assume(alice != address(0));
49        vm.assume(bob != address(0));
50        vm.assume(ceasar != address(0));
51
52        vm.assume(notBuiltinAddress(alice));
53        vm.assume(notBuiltinAddress(bob));
54        vm.assume(notBuiltinAddress(ceasar));
55
56        vm.assume(alice != bob);
57        vm.assume(alice != ceasar);
58        vm.assume(bob != ceasar);
59
60        vm.assume((amount >= token.balanceOf(bob)));
61        vm.assume((amount + token.balanceOf(bob)) <= (token.balanceOf(alice) + token.
62            balanceOf(bob)));

```

```

61 vm.assume((token.balanceOf(alice) + token.balanceOf(bob)) <= token.totalSupply());
62
63 uint256 oldBalanceAlice = token.balanceOf(alice);
64 uint256 oldBalanceBob = token.balanceOf(bob);
65 uint256 oldBalanceCeasar = token.balanceOf(ceasar);
66 uint256 oldTotalSupply = token.totalSupply();
67
68 vm.prank(alice);
69 token.transfer(bob, amount);
70
71 assertEquals(token.balanceOf(alice), oldBalanceAlice - amount);
72 assertEquals(token.balanceOf(bob), oldBalanceBob + amount);
73 assertEquals(token.balanceOf(ceasar), oldBalanceCeasar);
74
75 assertEquals(token.balanceOf(alice) + token.balanceOf(bob), oldBalanceAlice +
76     oldBalanceBob);
77 assertEquals(oldTotalSupply, token.totalSupply());
78
79 assertLe((token.balanceOf(alice) + token.balanceOf(bob)), token.totalSupply());
80 }
81
82 function testEqualTransferFrom(address alice, address ceasar, uint256 amount)
83     public {
84     kevm.infiniteGas();
85
86     vm.assume(alice != address(0));
87     vm.assume(ceasar != address(0));
88
89     vm.assume(notBuiltinAddress(alice));
90     vm.assume(notBuiltinAddress(ceasar));
91
92     vm.assume(alice != ceasar);
93
94     vm.assume((amount >= token.allowance(alice, alice)));
95     vm.assume(token.balanceOf(alice) <= token.totalSupply());
96
97     uint256 oldBalanceAlice = token.balanceOf(alice);
98     uint256 oldBalanceCeasar = token.balanceOf(ceasar);
99     uint256 oldTotalSupply = token.totalSupply();
100
101     vm.prank(alice);
102     token.transferFrom(alice, alice, amount);
103
104     assertEquals(token.balanceOf(alice), oldBalanceAlice);
105     assertEquals(token.balanceOf(ceasar), oldBalanceCeasar);
106
107     assertEquals(oldTotalSupply, token.totalSupply());
108
109     assertLe(token.balanceOf(alice), token.totalSupply());
110 }
111
112 function testTransferFrom(address alice, address bob, address ceasar, address david
113     , uint256 amount) public {
114     kevm.infiniteGas();
115
116     vm.assume(alice != address(0));
117     vm.assume(bob != address(0));
118     vm.assume(ceasar != address(0));
119     vm.assume(david != address(0));
120
121     vm.assume(notBuiltinAddress(alice));

```

```

120     vm.assume(notBuiltinAddress(bob));
121     vm.assume(notBuiltinAddress(ceasar));
122     vm.assume(notBuiltinAddress(david));
123
124     vm.assume(alice != bob);
125     vm.assume(alice != ceasar);
126     vm.assume(bob != ceasar);
127     vm.assume(bob != david);
128     vm.assume(ceasar != david);
129
130     vm.assume((amount >= token.allowance(alice, david)));
131     vm.assume((amount >= token.balanceOf(bob)));
132     vm.assume((amount + token.balanceOf(bob)) <= (token.balanceOf(alice) + token.
        balanceOf(bob)));
133     vm.assume((token.balanceOf(alice) + token.balanceOf(bob)) <= token.totalSupply());
134
135     uint256 oldBalanceAlice = token.balanceOf(alice);
136     uint256 oldBalanceBob = token.balanceOf(bob);
137     uint256 oldBalanceDavid = token.balanceOf(david);
138     uint256 oldBalanceCeasar = token.balanceOf(ceasar);
139     uint256 oldTotalSupply = token.totalSupply();
140
141     vm.prank(david);
142     token.transferFrom(alice, bob, amount);
143
144     assertEq(token.balanceOf(alice), oldBalanceAlice - amount);
145     assertEq(token.balanceOf(bob), oldBalanceBob + amount);
146     if (alice != david) {
147         assertEq(token.balanceOf(david), oldBalanceDavid);
148     }
149     assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
150
151     assertEq(token.balanceOf(alice) + token.balanceOf(bob), oldBalanceAlice +
        oldBalanceBob);
152     assertEq(oldTotalSupply, token.totalSupply());
153
154     assertLe((token.balanceOf(alice) + token.balanceOf(bob)), token.totalSupply());
155 }
156
    
```

A.3.8 PandoraFullUnchecked.t.sol

```

1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.13;
3
4 import "../src/PandoraUnchecked.sol";
5 import "forge-std/Test.sol";
6 import "kontrol-cheatcodes/KontrolCheats.sol";
7
8 contract HumanStandardTokenUncheckedTest is Test, KontrolCheats {
9     uint256 constant MAX_INT = 2**256 - 1;
10     HumanStandardTokenUnchecked token; // Contract under test
11
12     function setUp() public {
13         token = new HumanStandardTokenUnchecked(1000000000000000, "Pandora", 4, "PD");
14     }
15
16     function testEqualTransfer(address alice, address ceasar, uint256 amount) public {
17         kevm.infiniteGas();
18     }
19
    
```

```

19  vm.assume(alice != address(0));
20  vm.assume(ceasar != address(0));
21
22  vm.assume(notBuiltinAddress(alice));
23  vm.assume(notBuiltinAddress(ceasar));
24
25  vm.assume(alice != ceasar);
26
27  vm.assume(token.balanceOf(alice) <= token.totalSupply());
28
29  uint256 oldBalanceAlice = token.balanceOf(alice);
30  uint256 oldBalanceCeasar = token.balanceOf(ceasar);
31  uint256 oldTotalSupply = token.totalSupply();
32
33  vm.prank(alice);
34  token.transfer(alice, amount);
35
36  assertEq(token.balanceOf(alice), oldBalanceAlice);
37  assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
38
39  assertEq(oldTotalSupply, token.totalSupply());
40
41  assertLe(token.balanceOf(alice), token.totalSupply());
42  }
43
44  function testTransfer(address alice, address bob, address ceasar, uint256 amount)
45    public {
46    kevm.infiniteGas();
47
48    vm.assume(alice != address(0));
49    vm.assume(bob != address(0));
50    vm.assume(ceasar != address(0));
51
52    vm.assume(notBuiltinAddress(alice));
53    vm.assume(notBuiltinAddress(bob));
54    vm.assume(notBuiltinAddress(ceasar));
55
56    vm.assume(alice != bob);
57    vm.assume(alice != ceasar);
58    vm.assume(bob != ceasar);
59
60    vm.assume((amount >= token.balanceOf(bob)));
61    vm.assume((amount + token.balanceOf(bob)) <= (token.balanceOf(alice) + token.
62      balanceOf(bob)));
63    vm.assume((token.balanceOf(alice) + token.balanceOf(bob)) <= token.totalSupply());
64
65    uint256 oldBalanceAlice = token.balanceOf(alice);
66    uint256 oldBalanceBob = token.balanceOf(bob);
67    uint256 oldBalanceCeasar = token.balanceOf(ceasar);
68    uint256 oldTotalSupply = token.totalSupply();
69
70    vm.prank(alice);
71    token.transfer(bob, amount);
72
73    assertEq(token.balanceOf(alice), oldBalanceAlice - amount);
74    assertEq(token.balanceOf(bob), oldBalanceBob + amount);
75    assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
76
77    assertEq(token.balanceOf(alice) + token.balanceOf(bob), oldBalanceAlice +
78      oldBalanceBob);
79    assertEq(oldTotalSupply, token.totalSupply());
80  }

```

```

78     assertLe((token.balanceOf(alice) + token.balanceOf(bob)), token.totalSupply());
79     }
80
81
82     function testEqualTransferFrom(address alice, address ceasar, uint256 amount)
83         public {
84         kevm.infiniteGas();
85
86         vm.assume(alice != address(0));
87         vm.assume(ceasar != address(0));
88
89         vm.assume(notBuiltinAddress(alice));
90         vm.assume(notBuiltinAddress(ceasar));
91
92         vm.assume(alice != ceasar);
93
94         vm.assume((amount >= token.allowance(alice, alice)));
95         vm.assume(token.balanceOf(alice) <= token.totalSupply());
96
97         uint256 oldBalanceAlice = token.balanceOf(alice);
98         uint256 oldBalanceCeasar = token.balanceOf(ceasar);
99         uint256 oldTotalSupply = token.totalSupply();
100
101         vm.prank(alice);
102         token.transferFrom(alice, alice, amount);
103
104         assertEq(token.balanceOf(alice), oldBalanceAlice);
105         assertEq(token.balanceOf(ceasar), oldBalanceCeasar);
106
107         assertEq(oldTotalSupply, token.totalSupply());
108
109         assertLe(token.balanceOf(alice), token.totalSupply());
110     }
111
112     function testTransferFrom(address alice, address bob, address ceasar, address david
113         , uint256 amount) public {
114         kevm.infiniteGas();
115
116         vm.assume(alice != address(0));
117         vm.assume(bob != address(0));
118         vm.assume(ceasar != address(0));
119         vm.assume(david != address(0));
120
121         vm.assume(notBuiltinAddress(alice));
122         vm.assume(notBuiltinAddress(bob));
123         vm.assume(notBuiltinAddress(ceasar));
124         vm.assume(notBuiltinAddress(david));
125
126         vm.assume(alice != bob);
127         vm.assume(alice != ceasar);
128         vm.assume(bob != ceasar);
129         vm.assume(bob != david);
130         vm.assume(ceasar != david);
131
132         vm.assume((amount >= token.allowance(alice, david)));
133         vm.assume((amount >= token.balanceOf(bob)));
134         vm.assume((amount + token.balanceOf(bob)) <= (token.balanceOf(alice) + token.
135             balanceOf(bob)));
136         vm.assume((token.balanceOf(alice) + token.balanceOf(bob)) <= token.totalSupply());

```

```
137 uint256 oldBalanceDavid = token.balanceOf(david);
138 uint256 oldBalanceCeasar = token.balanceOf(ceasar);
139 uint256 oldTotalSupply = token.totalSupply();
140
141 vm.prank(david);
142 token.transferFrom(alice, bob, amount);
143
144 assertEquals(token.balanceOf(alice), oldBalanceAlice - amount);
145 assertEquals(token.balanceOf(bob), oldBalanceBob + amount);
146 assertEquals(token.balanceOf(ceasar), oldBalanceCeasar);
147 if (alice != david) {
148     assertEquals(token.balanceOf(david), oldBalanceDavid);
149 }
150
151 assertEquals(token.balanceOf(alice) + token.balanceOf(bob), oldBalanceAlice +
152     oldBalanceBob);
153 assertEquals(oldTotalSupply, token.totalSupply());
154
155 assertEquals((token.balanceOf(alice) + token.balanceOf(bob)), token.totalSupply());
156 }
```