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

Clara Schneidewind
TU Wien
Vienna, Austria
clara.schneidewind@tuwien.ac.at

Ilya Grishchenko
TU Wien
Vienna, Austria
ilya.grishchenko@tuwien.ac.at

Markus Scherer
TU Wien
Vienna, Austria
markus.scherer@tuwien.ac.at

Matteo Maffei
TU Wien
Vienna, Austria
matteo.maffei@tuwien.ac.at

ABSTRACT

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

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

ACM Reference Format:
Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 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 (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3372297.3417250

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417250

1 INTRODUCTION

Smart contracts introduced a radical paradigm shift in distributed computation, promising security in an adversarial setting thanks to the underlying consensus algorithm. Software developers can implement sophisticated distributed, transaction-based computations by leveraging the scripting language offered by the underlying blockchain technology. While many cryptocurrencies have an intentionally limited scripting language (e.g., Bitcoin [39]), Ethereum was designed from the ground up with a quasi Turing-complete language[1]. Ethereum smart contracts have thus found a variety of appealing use cases, such as auctions [25], data management systems [8], financial contracts [13], elections [38], trading platforms [37, 41], permission management [11] and verifiable cloud computing [18], just to mention a few. Given their financial nature, bugs and vulnerabilities in smart contracts may lead to catastrophic consequences. For instance, the infamous DAO vulnerability [1] recently led to a 60M$ financial loss and similar vulnerabilities occur on a regular basis [2, 3]. Furthermore, many smart contracts in the wild are intentionally fraudulent, as highlighted in a recent survey [10]. Even worse, due to the unmodifiable nature of blockchains, bugs or vulnerabilities in deployed smart contracts cannot be fixed.

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

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

1.1 State-of-the-art in Smart Contract Analysis

Existing approaches to smart contract analysis can be mainly classified as interactive frameworks for semantic-based machine-checked proofs [9, 12, 23, 27, 28, 47] and automated, heuristic-driven bug-finding tools [21, 33, 36, 40, 48]. Some recent works try to fill the middle ground between these two approaches, aiming at the best of the two worlds, i.e., an automated, yet sound static analysis of Ethereum smart contracts that can prove generic security properties [22, 32, 35, 46]. We conducted a thorough investigation, finding out that all of them fail to provide the intended soundness guarantees, which showcases the difficulty of this task. For details we refer the reader to [43]. Inspired by the issues that we see in the state of the art, we introduce a principled approach to the design and implementation of a sound, yet performant, static analysis tool for EVM bytecode.

1.2 Our Contributions

The contributions of this work can be summarized as follows:

- We design the first provably sound static analyzer for EVM bytecode, which builds on top of a Horn-clause-based reachability analysis. We show that reachability analysis suffices to verify interesting security properties for smart contracts as well as contract-specific functional properties via an encoding into Hoare-style reasoning. The design of such static analysis is technically challenging, since it requires careful abstractions of various EVM components (e.g., the stack-based execution model, the gas bounding the smart contract execution, and the memory model) as well as a dedicated over-approximation of blockchain data which are not statically known and yet affect contract execution (e.g., the code of other contracts which may act both as callers and callees);
- We prove our static analysis technique sound against the formal semantics of EVM bytecode by Grishchenko et al. [23];
- In order to facilitate future refinements of our analysis, as well as the design of similar static analyses for other languages, we design and implement *HoRSt*, a framework for the specification and implementation of static analyses based on Horn clause resolution. Specifically, *HoRSt* takes as input a (mathematical) specification of the Horn clauses defining the static analysis and produces an `smt-lib` [7] encoding suitable for *z3* [29], which includes various optimizations such as Horn clause and constant folding;
- We use *HoRSt* to implement the static analyzer *eThor*. To gain confidence in the resulting implementation, we encode the relevant semantic tests (604 in total) of the official EVM suite as reachability properties, against which we successfully test the soundness and precision of *eThor*;
- We conduct a large-scale experimental evaluation on real-world contracts comparing *eThor* to the state-of-the-art analyzer ZEUS [32] which claims to provide soundness guarantees. While ZEUS shows a striking specificity (i.e., completeness) of 99.8%, *eThor* clearly outperforms ZEUS in terms of recall (i.e., soundness) – 100% vs. 11.4% – which empirically refutes ZEUS' soundness claim. With a specificity of 80.4%, *eThor* shows an overall performance of 89.1% (according to the F-measure) as compared to ZEUS' F-measure of 20.4%.

The remainder of this paper is organized as follows. § 2 reviews Ethereum and the semantics of EVM bytecode. § 3 introduces our static reachability analysis, specifies its soundness guarantee and discusses relevant smart contract properties in scope of the analysis. § 4 introduces the specification language *HoRSt*. § 5 presents *eThor* and our experimental evaluation. § 6 overviews related work. § 7 concludes by discussing interesting future research directions. The appendix provides additional material. The source code of *eThor* and *HoRSt* with the data set used in the experimental evaluation are available online [6].

2 ETHEREUM

We first introduce background on Ethereum (§ 2.1) and then overview the EVM bytecode semantics by [23] utilized in this work (§ 2.2).

2.1 Background

The Ethereum platform can be seen as a transaction-based state machine where transactions alter the global state of the system, which consists of accounts. There are two types of accounts: External accounts, which are owned by a user of the system, and contract accounts, which can be seen as distributed programs. All accounts hold a balance in the virtual currency *Ether*. Additionally, contract accounts include persistent storage and the contract's code. Transactions can either create new contract accounts or call existing accounts. Calls to external accounts can only transfer Ether to this account, but calls to contract accounts additionally execute the account's contract code. The contract execution might influence the storage of the account and might as well perform new transactions – in this case, we speak of *internal transactions*. The effects of contract executions are determined by the *Ethereum Virtual Machine* (EVM). This virtual machine characterizes the *quasi Turing complete* execution model of Ethereum smart contracts where the otherwise Turing complete execution is restricted by an upfront defined resource *gas* that effectively limits the number of execution steps. A transaction's originator can specify an upper bound on the gas that she is willing to pay for the contract execution and also sets the gas price (the amount of Ether to pay for a unit of gas). The originator then prepays the specified gas limit and gets refunded according to the remaining gas in case of successful contract execution.

EVM bytecode. Contracts are published on the blockchain in form of *EVM bytecode*– an Assembler like bytecode language. The EVM is a stack-based machine and specifies the semantics of bytecode instructions. Consequently, EVM bytecode mainly consists of standard instructions for stack operations, arithmetics, jumps and local memory access. The instruction set is complemented with blockchain-specific instructions such as an opcode for the SHA3 hash and several opcodes for accessing information on the current (internal) transaction. In addition, there are opcodes for accessing and modifying the storage of the executing account and distinct opcodes for initiating internal transactions.

Each instruction is associated with (a potentially environment-dependent) gas cost. If the up-front defined gas-limit is exceeded during execution, the transaction execution halts exceptionally and the effects of the current transaction on the global state are reverted. For nested transactions, an exception only reverts the effects of the executing transaction, but not those of the calling transactions.

$$
\begin{array}{rrcl}
\text{Callstacks} & S & := & EXC::U \mid HALT(\sigma, gas, d, \eta)::U \mid U \\
\text{Plain callstacks} & U & := & (\mu, \iota, \sigma, \eta)::U \mid \epsilon \\
\text{Machine states} & \mu & := & (gas, pc, m, i, s) \\
\text{Account states} & a & := & (n, b, code, stor)
\end{array}
$$

Figure 1: Grammar for calls stacks

Solidity. In practice, Ethereum smart contracts are shipped and executed in EVM bytecode format but are, for a large part, written in the high-level language Solidity [5]. The syntax of Solidity resembles JavaScript, enriched with primitives accounting for the distributed blockchain setting. Solidity exhibits specific features that give rise to smart contract vulnerabilities, as will be discussed in § 2.3. We will not give a full account of Solidity's language features here, but add explanations throughout the paper when needed.

2.2 EVM Semantics

Our static analysis targets a recently introduced small-step semantics for EVM bytecode [23], which we shortly review below.[2]

The semantics of EVM bytecode is given by a small-step relation $\Gamma \vDash S \rightarrow S'$ that encompasses the possible steps that a callstack S, representing the overall state of a contract execution, can make under the transaction environment Γ. The transaction environment Γ summarizes static information about the transaction execution such as the information on the block that the transaction is part of and transaction-specific information such as gas price or limit. We write $\Gamma \vDash S \rightarrow^* S'$ for the reflexive transitive closure of the small-step relation and call the pair (Γ, S) a *configuration*.

Configurations. The most important formal components of EVM configurations are summarized in Figure 1.

Global State. Ethereum's global state σ is given as a mapping from account addresses to account states. An account state consists of a nonce n that is incremented in the course of contract creation, a balance b, a persistent storage *stor*, and the account's *code*. External accounts have no code and hence cannot access storage.

Callstacks. The overall state of an external transaction is captured by a stack of execution states that we will refer to as *callstack*. The individual execution states reflect the states of the pending internal transactions. More formally, the elements of a callstack are either regular execution states of the form $(\mu, \iota, \sigma, \eta)$ or terminal execution states $HALT(\sigma, gas, d, \eta)$ and EXC which can only occur as top stack elements. For terminated executions we differentiate between exceptional halting (EXC), which will revert all effects of the transaction, and regular halting $HALT(\sigma, gas, d, \eta)$, in which case the effects of the transaction are captured by the global state σ at the point of halting, the *gas* remaining from the execution, the return data d, and the transaction effects η (effects that will only be applied after completing the external execution).

The state of a non-terminated internal transaction is described by a regular execution state of the form $(\mu, \iota, \sigma, \eta)$. During execution, this state tracks the current global state σ of the system, the execution environment ι to the internal transaction (which specifies parameters such as the input to the transaction and the code to be executed) as well as the local state μ of the stack machine, and the transaction effects η that will be applied after transaction execution.

The local machine state μ reflects the state of the stack machine that handles local computations. It is represented by a tuple (gas, pc, m, i, s) holding the amount of *gas* available for execution, the program counter *pc*, the local memory m, the number of active words in memory i, and the machine stack s. As the stack machine models local computations, the execution of every new (internal) transaction starts again in a fresh machine state at program counter zero with an empty stack and zero-initialized memory. Only the gas value is initialized as specified by the initiator of the transaction.

Small-step Rules. We illustrate the mechanics of the EVM bytecode semantics by an example and refer to [23] for a full definition.

Local instructions, e.g., ADD, only operate on the machine state:

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

Given a stack that contains at least two values and given a sufficient amount of gas (here 3 units), an ADD instruction takes two values from the stack and pushes their sum. These effects, as well as the advancement in the program counter and the substraction of the gas cost, are reflected in the updated machine state μ'.

A more evolved semantics is exhibited by the transaction initiating instructions (CALL, CALLCODE and DELEGATECALL, CREATE). Intuitively, CALL executes the callee's code in its own environment, CALLCODE executes the callee's code in the caller's environment, which might be useful to call libraries implemented in a separate contract, and DELEGATECALL goes a step further by preserving not only the caller's environment but even part of the environment of the previous call (e.g., the sender information), which effectively treats the callee's code as an internal function of the caller's code. Finally, the CREATE instruction initiates an internal transaction that creates a new account. Instructions from this set are particularly difficult to analyze, since their arguments are dynamically evaluated and the execution environment has to be tracked and properly modified across different calls. Further, it can well be that the code of a called function is not accessible at analysis time, e.g., because the contract allows for money transfers to a dynamic set of contracts (as the DAO contract discussed in the next section).

2.3 Security Properties of Smart Contracts

Ethereum smart contracts have undergone several severe attacks in the past that were enabled by major bugs in the contract code, most prominently the DAO hack [1]. Interestingly, this bug can be traced back to the violation of a generic security property of the attacked contract, called single-entrancy. We will shortly present the class of reentrancy attacks and the underlying security property.

Preliminary Notions. In order to present security properties in a concise fashion, the previously presented small-step semantics is augmented with an annotation to callstack elements that reflects the currently executed contract. We say that an execution state s is *strongly consistent* with annotated contract c if s executes c (according to the execution environment) and c is present in the global state of s. Further, for arguing about EVM bytecode executions, we

[2] Recent additions to the EVM semantics such as STATICCALL, CREATE, and CREATE2, are not explicitly mentioned here, but covered by our static analysis specified in [6].

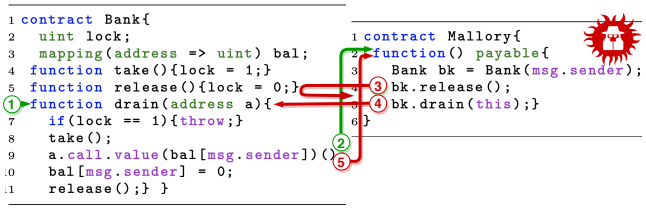

Figure 2: Reentrancy Attack.

are only interested in those initial configurations that might result from a valid external transaction in a valid block. We call these configurations *reachable* and refer to [23] for a detailed definition.

Single-entrancy. Intuitively, single-entrancy characterizes the robustness of a contract against reentrancy attacks [10, 36]. Reentrancy attacks exploit that a contract calling another contract can be called back (reentered) before completing the original internal transaction. At the point of reentering the contract can then be in an inconsistent state which allows for unintended behavior. In the DAO hack, the attacker stole all funds of a contract by reentering the contract and sending money to herself. We exemplify this kind of attack by the `Bank` contract in Figure 2: this has a basic reentrancy protection in place which however can easily be circumvented.

The `Bank` contract implements a simple banking functionality, keeping the balance of all users (identified by their addresses) in the mapping `bal`. We only discuss the contract function `drain` which allows a user to transfer all its money from its bank account to the provided beneficiary address `a`. For protecting against reentrancy, the `drain` function implements a locking functionality: it is only entered in case the lock is not taken. In this case it takes the lock (using function `take`), transfers the remaining balance of the function callee (denoted by `msg.sender`) to the beneficiary address `a`, updates the user's balance, and releases the lock. Note that the `call` construct (being translated to a CALL instruction in EVM bytecode) does not only trigger the value transfer, but also invokes the execution of the callee's so-called *fallback function* (written as a function without name or arguments in Solidity as depicted in the `Mallory` contract in Figure 2). Hence, the use of a `call` can cause the executed contract to be reentered during execution, potentially undermining the contract integrity. The locking mechanism should prevent this problem by causing an exception in case the contract is reentered (indicated by the lock being taken). However, since the locking functionality is publicly accessible, a reentrancy attack (as depicted in Figure 2) is possible: An attacker calling the `drain` function (via `Mallory`) with `Mallory`'s address as argument (①) transfers all of `Mallory`'s funds back to her and executes her fallback function (②). `Mallory` then first calls the public `release` function to release the lock (③) and next calls the `drain` function of `Bank` again (④). Since the attacker's balance has not been updated at this point, another transfer to `Mallory` can be performed (⑤). These steps can be continued until running out of gas or reaching the callstack limit. In both cases the last value transfer is rolled back, but the effects of all former internal transactions persist, leaving the contract `Bank` drained-out.

The security property ruling out these attacks is called *single-entrancy* and is formalized below. Intuitively, a contract is single-entrant if it cannot perform any more calls after reentering.

Definition 2.1 (Single-entrancy [23]). A contract c is single-entrant if for all reachable configurations $(\Gamma, s_c{::}S)$, for all s', s'', S'

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

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

Single-entrancy is particularly interesting in that it constitutes a generic robustness property of smart contracts. Other prominent vulnerabilities [2, 3] are caused by functional correctness issues particular to a specific contract. For spotting such issues, contract-specific correctness properties need to be defined and verified. We discuss the formalization of such properties in § 3.5.

3 STATIC ANALYSIS OF EVM BYTECODE

Starting from the small-step semantics presented in § 2.2, we design a sound reachability analysis that supports (among others) the validation of the single-entrancy property. We follow the verification chain depicted in Figure 3: For showing the executions of a contract to satisfy some property Φ, we formulate a *Horn-clause-based abstraction* that abstracts the contract execution behavior and argue about an *abstracted property* over *abstract executions* instead. This reasoning is sound given that all concrete small-step executions are modeled by some abstract execution and given that the abstracted property over-approximates Φ.

A Horn-clause-based abstraction for a small-step semantics $\longrightarrow$ is characterized by an abstraction function α that translates concrete configurations (here ●) into *abstract configurations* (here ▲). Abstract configurations are sets of predicate applications where predicates (formally characterized by their signature $\mathcal{S}$) range over the values from abstract domains. These abstract arguments are equipped with an order $\leq$ that can be canonically lifted to predicates and further to abstract configurations, hence establishing a notion of precision on the latter. Intuitively, α translates a concrete configuration into its most precise abstraction. The *abstract semantics* is specified by a set of Constrained Horn clauses Λ over the predicates from $\mathcal{S}$ and describes how abstract configurations evolve during abstract execution. A Constrained Horn clause is a logical implication that can be interpreted as an inference rule for a predicate, consequently an abstract execution consists of logical derivations from an abstract configuration using Λ. A Horn-clause-based abstraction constitutes a sound approximation of small-step semantics $\longrightarrow$ if every concrete (multi-step) execution $● \longrightarrow^* ●'$ can be simulated by an abstract execution: More precisely, from the abstract configuration $\alpha(●)$ one can logically derive using Λ an abstract configuration ▲ that constitutes an over-approximation of $●'$ (so is at least as abstract as $\alpha(●')$). A formal presentation of the soundness statement is given in § 3.4 while a characterization in abstract interpretation terminology is deferred to § B.2. A sound abstraction allows for the provable analysis of *reachability properties*: Such properties can be expressed as sets of problematic configurations (here ●). Correspondingly, a sound abstraction for such a

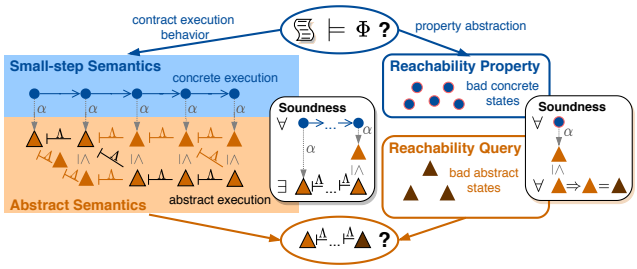

Figure 3: Formal verification chain of *eThor*. $\Delta \vdash^{\Lambda} \Delta'$ denotes that the abstract configuration Δ' can be logically derived from Δ' (within one step) using the Horn clauses in Λ.

property is a set of bad abstract configurations (here ▲) which contains all possible over-approximations of the bad concrete states. The soundness of the abstract semantics then guarantees that if no bad abstract configuration from this set can be entered, also no bad configuration can be reached in the concrete execution.

3.1 Main Abstractions

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

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

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

Memory model. In the EVM the local memory is byte-indexed, while the machine stack holds words (encompassing 32 bytes). Consequently loading a machine word from memory requires to assemble the byte values from 32 consecutive memory cells. However, as already described in [42], in practice reasoning about this conversion between words and bytes is hard. Therefore, we model memory in our abstraction as a word array: this enables very cheap accesses in case that memory is accessed at the start of a new memory word, and otherwise just requires the combination of two memory words.

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

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

Figure 4: Definition of the predicate signature $\mathcal{S}_{c^*}$ and the abstract domain $\hat{D}$.

3.2 Analysis Definition

In the following we formally specify our analysis by defining the underlying Horn-clause-based abstraction. An abstract configuration is a set of predicate applications representing one or several concrete configurations. Since we are interested in analyzing executions of the contract c^*, we consider EVM configurations representing such executions which are call stacks having an execution state of contract c^* as a bottom element. We abstract such a call stack by the set of all its elements that describe executions of c^*, reflecting the stack structure only by indicating whether a relevant execution state represents the original execution of c^* (call level 0) or a reentering execution that hence appears higher on the call stack (call level 1). The individual execution states are abstracted as predicate applications using the predicates listed in Figure 4: A predicate application of the form $\mathrm{MState}_{\mathrm{pc}}((\mathit{size}, s), m, \mathit{stor}, c\ell)$ describes a regular execution of c^* at program counter pc that has a local stack of size *size* with elements as described by the mapping s (from stack positions to elements) and a local memory m, and the global storage of contract c^* at this point being *stor*. Accordingly, the predicate application $\mathrm{Exc}(c\ell)$ denotes that an execution of c^* halted exceptionally on call level $c\ell$ and $\mathrm{Halt}(\mathit{stor}, c\ell)$ represents an execution that halted regularly on call level $c\ell$ with the global storage of c^* being *stor*. Since during the abstract execution, a precise modeling of all the described state components is not always possible, the argument domains of the predicates encompass the abstract domain $\hat{D}$ that enriches $\mathbb{N}$ with the join element $\top$ over-approximating any natural number. Formally, the described abstractions of EVM configurations are captured by the abstraction function α in Figure 5 that maps call stacks into the corresponding sets of predicates, yielding an abstract configuration.

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

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

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

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

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

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

Figure 5: Configuration abstraction function. Here $v||_n w$ denotes the value obtained by concatenating v's and w's byte representation, assuming that w is represented by n bytes.

$$\begin{aligned}
&(\!|\mathsf{ADD}|\!)_{\mathsf{pc}} := \\
&\quad \{\, \mathsf{MState}_{\mathsf{pc}}((size, s), m, stor, c\ell) \ \wedge \ size > 1 \\
&\quad\ \wedge \ \hat{x} = s[size - 1] \ \wedge \ \hat{y} = s[size - 2] \\
&\quad\ \Longrightarrow \ \mathsf{MState}_{\mathsf{pc}+1}((size - 1, s[size - 2 \rightarrow \hat{x} \mathbin{\widehat{+}} \hat{y}]), m, stor, c\ell), \quad \text{(A1)} \\
&\quad \mathsf{MState}_{\mathsf{pc}}((size, s), m, stor, c\ell) \ \Longrightarrow \ \mathsf{Exc}(c\ell) \,\} \quad \text{(A2)} \\
&(\!|\mathsf{CALL}|\!)_{\mathsf{pc}} := \\
&\quad \{\, \mathsf{MState}_{\mathsf{pc}}((size, s), m, stor, c\ell) \ \wedge \ size > 6 \\
&\quad\ \Longrightarrow \ \mathsf{MState}_{\mathsf{pc}+1}((size - 6, s[size - 7 \rightarrow \top]), \lambda x.\,\top, \lambda x.\,\top, c\ell), \quad \text{(C1)} \\
&\quad \mathsf{MState}_{\mathsf{pc}}((size, s), m, stor, c\ell) \ \wedge \ size > 6 \\
&\quad\ \Longrightarrow \ \mathsf{MState}_0((0, \lambda x.\,0), \lambda x.\,0, stor, 1), \quad \text{(C2)} \\
&\quad \mathsf{MState}_{\mathsf{pc}}((size, s), m, stor, c\ell) \ \wedge \ size > 6 \ \wedge \ \mathsf{Halt}(stor_h, 1) \\
&\quad\ \Longrightarrow \ \mathsf{MState}_0((0, \lambda x.\,0), \lambda x.\,0, stor_h, 1), \ldots\} \quad \text{(C3)}
\end{aligned}$$

Figure 6: Partial definition of $(\!|\cdot|\!)_{\mathbf{pc}}$: selection of abstract semantics rules. For CALL the exception rule is omitted.

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

The core of the abstract semantics is defined by the *instruction abstraction function* $(\!|\cdot|\!)_i$ that maps a contract instruction at position i to a set of Horn clauses over-approximating the semantics of the corresponding instruction. We will discuss the translation of the ADD and CALL instruction depicted in Figure 6 to illustrate the main features of the abstract semantics.

Addition. The abstract semantics of the addition instruction (ADD) encompasses two Horn clauses describing the successful execution and the failure case. A prerequisite for a successful addition is the existence of a sufficient amount of arguments on the machine stack. In this case, the top stack values are extracted and the stack at the next program counter (modeled by the predicate $\mathsf{MState}_{\mathsf{pc}+1}$) is updated with their sum. As the stack elements, however, range over the abstract value domain $\hat{D}$, the addition operation on $\mathbb{N}$ needs to be lifted to $\hat{D}$: Following the general intuition of $\top$ representing all potential values in $\mathbb{N}$, the occurrence of $\top$ as one of the operands immediately declassifies the result to $\top$. Similar liftings are performed for all unary, binary and comparison operators in the instruction set. A precise definition is given in § B.3.

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

Contract Calls. The abstraction for CALL is the most interesting. This instruction takes seven arguments from the stack that specify parameters to the call, such as the target of the call or the value to be transferred, as well as the memory addresses specifying the location of the input and the return data. When returning from a successful contract call, the value 1 is written to the stack and the return value is written to the specified memory fragment. The persistent storage after a successful call contains all changes that were performed during the execution of the called contract. In the case of an exception the storage is rolled back to the point of calling and the value 0 is written to the stack to indicate failure.

Since a contract CALL initiates the execution of another (unknown) contract, all its effects on the executions of c^* need to be modeled. More precisely, these effects are two-fold: the resuming execution of c^* on the current call level needs to be approximated, as well as the reentering executions of c^* (on a higher call level). For obtaining an analysis that is precise enough to detect real-world contracts with reentrancy protection as secure, it is crucial to model c^*'s persistent storage as accurately as possible in reentering executions. This makes it necessary to carefully study how the storage at the point of reentering relates to the one in the previous executions of c^*, taking into account that (in the absence of DELEGATECALL and CALLCODE instructions in c^*) only c^* can manipulate its own storage. Figure 7 overviews the storage propagation in the case of a contract call: To this end it shows the sequence diagram of a concrete execution of c^* that calls a contract c' which again triggers several reentrancies of c^*. In this course three ways of storage propagation between executions of c^* are exhibited: 1) The storage is *forward propagated* from a calling execution to a reentering execution of c^* (A, C) 2) The storage is *cross propagated* from a finished reentering execution to another reentering execution of c^* (B) 3) The storage is *back propagated* from a finished reentering execution to a calling execution of c^* (D, E) These three kinds of propagation are reflected in the three abstract rules for the call instruction given in Figure 6 and correspondingly visualized in Figure 7.

Rule (C1) describes how the execution of c^* (original and reentering alike) resumes after returning from the call, and hence approximates storage back propagation: For the sake of simplicity, storage gets over-approximated in this case by $\lambda x.\,\top$. The same applies to the local memory and stack top value as those are affected by the result of the computation of the unknown contract. Rule (C2) captures the initiation of a reentering execution (at call level 1) with storage forward propagation: As contract execution always starts at program counter 0 with empty stack and all-zero local memory,

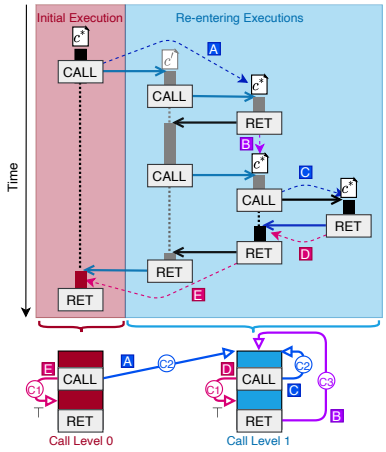

Figure 7: Illustration of the different call abstractions.

only abstractions (instances of the MState_0 predicate) of this shape are implied. The forward propagation of storage is modeled by initializing the MState_0 predicate with the storage *stor* at call time. Rule (C3) models storage cross propagation: Similar to rule (C2), an abstract reentering execution in a fresh machine state is triggered. However, the storage is not propagated from the point of calling, but from a finished reentering execution whose results are abstracted by the halting predicate Halt at call level 1. This rule is independent of the callee in that it is only conditioned on the reachability of some CALL instruction, but it does not depend on the callee's state. Its cyclic structure requires extrapolating an invariant on the potential storage modifications that are computable by c^*: Intuitively, when reentering c^* it needs to be considered that priorly the storage was modified by applying an arbitrary sequence of c^*'s public functions. The significance of this abstraction is motivated by the example in Figure 2 where the attack is only enabled by calling `Bank`'s `release` function first, to release the lock before reentering.

3.3 Scope of the analysis

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

Following this line of argumentation, we assume c^* not to contain DELEGATECALL and CALLCODE instructions: these instructions enable the execution of another contract code in the context of c^*, allowing for the modification of the persistent storage of c^* and even of money transfers on behalf of c^*. Using DELEGATECALL or CALLCODE to call an unknown contract can therefore potentially result in the reachability of any execution states of contract c^*. Consequently every property relying on the non-reachability of certain problematic contract states would be considered violated. In a setting of multiple known contracts the restriction on DELEGATECALL and CALLCODE instructions could be relaxed to allow for such calls that are guaranteed to target known contracts.

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

3.4 Soundness Result

We prove, for each contract c^*, that the defined Horn-clause-based abstraction soundly over-approximates the small-step semantics presented in § 2.2. Formally, this property is stated as follows:

Theorem 3.1 (Soundness). *Let c^* be a contract not containing DELEGATECALL or CALLCODE. Let Γ be a transaction environment and let S and S' be annotated callstacks such that $|S'| > 0$. Then for all execution states s that are strongly consistent with c^* it holds that*

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

The theorem states that every execution of contract c^* (modeled by a multi-step execution starting in state s_{c^*} on an arbitrary call stack S and ending in call stack $S' +\!\!+ S$, indicating that the original execution of c^* yielded the state as modeled by the call stack S'), can be mimicked by an abstract execution. This means that from every abstract configuration Δ_I that abstracts s_{c^*} (so that it is more abstract than $\alpha([s_{c^*}])$) one can logically derive using the Horn clauses in $\delta(c^*)$ some abstract configuration Δ abstracting S'. As a consequence of this theorem we can soundly reason about arbitrary executions of a contract c^*: if we can show that from an abstract configuration Δ_I, that abstracts a set of initial execution states of c^*, it is impossible to derive using $\delta(c^*)$ some other abstract configuration Δ, that abstracts a set of problematic execution states of c^*, then this ensures that all these problematic states are not reachable with a small-step execution from any of the initial states.

For the proof of Theorem 3.1 we refer the reader to the extended version of this paper at [6].

3.5 Reachability Properties for Contract Safety

As characterized by the soundness result, our abstraction allows for the sound analysis of reachability properties. We will illustrate

in the following how such a reachability analysis is sufficient to express relevant smart contract security properties.

Single-entrancy. Some generic security properties of Ethereum smart contracts can be over-approximated by reachability properties and thus checked automatically by our static analysis. Consider, the single-entrancy property from § 2.3 which has been proven to be approximated by the following reachability property in [22].

Definition 3.2 (Call unreachability [22]). A contract c is call unreachable if for all regular execution states $(\mu, \iota, \sigma, \eta)$ that are strongly consistent with c and satisfy $\mu = (g, 0, \lambda x.\, 0, 0, \epsilon)$ for some $g \in \mathbb{N}$, it holds that for all transaction environments Γ and all callstacks S

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

Where $Inst_{call} = \{\mathsf{CALL}, \mathsf{CALLCODE}, \mathsf{DELEGATECALL}, \mathsf{CREATE}\}$

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

Static assertion checking. The Solidity language supports the insertion of assertions into source code. Assertions shall function as pure sanity checks for developers and are enforced at runtime by the compiler creating the corresponding checks on the bytecode level and throwing an exception (using the INVALID opcode) in case of an assertion violation [19]. However, adding these additional checks creates a two-fold cost overhead: At create time a longer bytecode needs to be deployed (the longer the bytecode the higher the gas cost for creation) and at call time the additional checks need to be executed which results in additional gas consumption. With our static analysis technique, assertions can be statically checked by querying the reachability of the INVALID instruction. If no such instruction is reachable, by the soundness of the analysis, the code is proven to give the same guarantees as with the assertion checks (up to gas) and those checks can safely be removed from the code resulting in shorter and cheaper contracts.[3] Formally, we can characterize this property as the following reachability property:

Definition 3.3 (Static assertion checking). Let c be a contract and $(\mu, \iota, \sigma, \eta)$ regular execution states such that $(\mu, \iota, \sigma, \eta)$ is strongly consistent with c and $\mu = (g, 0, \lambda x.\, 0, 0, \epsilon)$ for some $g \in \mathbb{N}$. Let Γ be an arbitrary transaction environment and S be an arbitrary callstack. Then a the static assertion check for c is defined as follows:

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

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

Semi-automated verification of contract-specific properties. As demonstrated by Hildebrandt et al. [27], reachability analysis can be effectively used for Hoare-Logic-style reasoning. This holds in particular for the analysis tool presented in this work: Let us consider a Hoare triple $\{P\}C\{Q\}$ where P is the precondition (operating on the execution state), C is the contract code and Q is the postcondition that should be satisfied after executing code C in an execution state satisfying P. Then we can intuitively check this claim by checking that a state satisfying $\neg Q$ can never be reached when starting execution in a state satisfying P. More formally, we can define Hoare triples as reachability properties as follows:

Definition 3.4 (Hoare triples). Let c^* be a contract and let C be a code fragment of c^*. Let $P \in \mathcal{S} \rightarrow \mathbb{B}$ be a predicate on execution states (strongly consistent with c^*) that models execution right at the start of C and similarly let $Q \in \mathcal{S} \rightarrow \mathbb{B}$ be a predicate on execution states (strongly consistent with c^*) that models execution right at the point after executing C. Then Hoare triples $\{P\}C\{Q\}$ can be characterized as follows:

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

Hoare-Logic style reasoning can be used for the semi-automated verification of smart contracts given that their behavior is specified in terms of pre- and postconditions. For now it still requires a non-negligible amount of expertise to insert the corresponding abstract conditions on the bytecode-level, but by a proper integration into the Solidity compiler the generation of the initialization and reachability queries could be fully automated (cf. § C.1). We want to stress that in contrast to existing approaches, our analysis technique has the potential to provide fully automated pre- and postcondition checking even in the presence of loops as it leverages the fixed point engines of state-of-the-art SMT solvers [30].

4 *HORST*: A STATIC ANALYSIS LANGUAGE

To facilitate the principled and robust development of static analyzers based on Horn clause resolution, we designed *HoRSt* – a framework consisting of a high-level specification language for defining Horn-clause-based abstractions and a compiler generating optimized `smt-lib` encodings for SMT solvers. The objective of *HoRSt* is to assist analysis designers in developing fast and robust static analyzers from clean and readable logical specifications.

Many existing practical analyzers are built on top of modern SMT solvers such as *z3*. These solvers are highly optimized for performance, however they show big performance deviations on different problem instances which are (due to the heavy use of heuristics) difficult to predict for the users. Handcrafting logical specifications for such solvers in their low-level input format `smt-lib` is hence not only cumbersome, error-prone, and requires technical expertise, but is also very inflexible, since the performance effects of different encodings may vary with the concrete problem instance. For tackling this issue, *HoRSt* decouples the high-level analysis design from the compilation to the input format: A high-level specification format allows for clear, human-readable analysis definitions while the translation process is handled by a stable and streamlined backend. On top, *HoRSt* allows for easily applying and experimenting with different Horn-clause-level optimizations that we can show to enhance the performance of *z3* substantially in our problem domain. We will shortly illustrate the utilization of *HoRSt* in the design process of our static analyzer and discuss the most interesting optimizations performed by the *HoRSt* compiler. For an introduction to the *HoRSt* language, we refer the reader to § A.1.

[3] The Solidity Docs [19] discuss exactly this future use of static analysis tools for assertion checking.

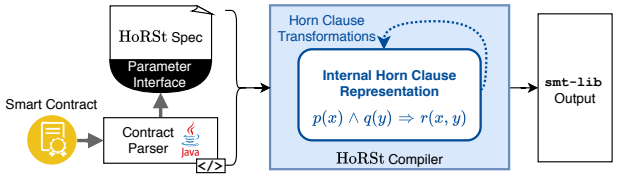

Figure 8: Utilization of *HoRSt* for static analysis

Designing static analyses using HoRSt. The *HoRSt* language allows for writing math-like specifications of Horn clauses such as those given in Figure 6. For parametrizing those clauses (e.g., by the program counters of a specific contract) an interface with a *Java*™ back end can be specified to handle the domain-specific infrastructure, such as contract parsing. We overview the different steps of the analysis design process in Figure 8.

The core of the analysis is the *HoRSt* specification. Using high-level programming constructs such as algebraic data types, pattern matching, and bounded iteration, a *HoRSt* specification describes Constrained Horn clauses over user-defined predicates. Horn clauses can be parametrized by (families) of sets that are specified in the parameter interface (e.g., the sets of all program counters containing a certain bytecode instruction in a specific contract). Given such a specification, the analysis designer needs to provide infrastructure code written in *Java*™. In particular this code needs to exhibit an implementation of those sets (or functions) specified in the parameter interface. In the case of our analysis, the environment code contains the infrastructure for contract parsing and the parameter interface allows for accessing the assembled contract information (code length, positions of opcodes, etc.) in the analysis specification. The *HoRSt* compiler itself is utilized to generate (optimized) `smt-lib` output given a *HoRSt* specification and the parameter interface implementation: It unfolds the high-level specification into separate Horn clauses over basic data types, applying the interface implementation: To this end it also resolves all high-level constructs, ensuring that the resulting Horn clauses fall into the fragment that can be handled by *z3*. On top, the *HoRSt* compiler (optionally) performs different optimizations and transformations on the resulting Horn clauses, before translating them into the standardized SMT output format `smt-lib`. The most important of these transformations are discussed in the following.

Low-level optimizations. One of the most effective optimizations performed by *HoRSt* is the predicate elimination by *unfolding* Horn clauses. This satisfiability preserving transformation has been long-studied in the literature [15, 44] and showed beneficial for solving Horn clauses in certain settings [14, 26]. In practice, however, the exhaustive application of this transformation can lead to an exponential blow-up in the number of Horn clauses and hence does not necessarily yield the best results. For this reason *HoRSt* implements different strategies for the (partial) application of this transformation, which we call *linear folding* and *exhaustive folding*.

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

5 IMPLEMENTATION & EVALUATION

We use *HoRSt* to generate the analyzer *eThor* which implements the static analysis defined in § 3. In the following, we overview the design of *eThor* and illustrate how *eThor* can enhance smart contract security in practice. To this end we conduct a case study on a widely used library contract, showing *eThor*'s capability of verifying functional correctness properties and static assertion checks. Further, we validate *eThor*'s soundness and precision on the official EVM test-suite and run a large-scale evaluation for the single-entrancy property on a set of real-world contracts from the Ethereum blockchain, comparing *eThor* with the state-of-the-art analyzer ZEUS [32].

5.1 Static Analysis Tool

The mechanics of *eThor* are outlined in Figure 9: *eThor* takes as input the smart contract to be analyzed in bytecode format and a *HoRSt*-specification parametrized by the contract. To enhance the tool's performance and precision, *eThor* performs a multi-staged analysis: First, it approximates the contract jump destinations. This step decouples the control flow reconstruction (which can be performed more efficiently with a less precise abstract semantics as typically no computations on jump destinations are performed, but just their flow during the stack needs to be modeled) from the more evolved abstract semantics required for precisely analyzing the properties discussed in § 3.5. As both used semantics are sound, the soundness of the overall analysis is guaranteed. In a second preprocessing step, *eThor* performs a simple partial execution of atomic program blocks in order to statically determine fixed stack values. This can be beneficial in order to, e.g., precompute hash values and results of exponentiation which would otherwise need to be over-approximated in the analysis due to the lacking support for such operations by *z3*. The results from the preanalysis steps are incorporated into the analysis by a predefined interface in the *HoRSt*-specification. The *HoRSt* compiler then – given the interface implementation and the specification – creates an internal Horn clause representation which, after optionally performing different optimizations, is translated to an `smt-lib` file on which the SMT solver *z3* is invoked. The reconstructed control flow is obtained by a *Soufflé* [31] program, which was created by manually translating a *HoRSt* specification. *Soufflé* is a high performance datalog engine, which we plan to support as a compilation target for (a subset of) *HoRSt* in the near future. Since the problem of control flow reconstruction falls into the fragment supported by modern datalog solvers, we found *Soufflé* more performant than using the general-purpose solver *z3* in this context [4]. However, for reasoning about more involved properties, the expressiveness of *z3* is required as we will illustrate in § 5.2.

5.2 Case Study: SafeMath Library

As a case study for functional correctness and assertion checking we chose Solidity's `SafeMath` library [4], a library implementing proper exception behavior for standard arithmetic operations. This particularly encompasses exceptions in case of overflows, underflows, and

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

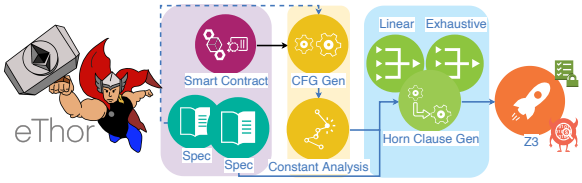

Figure 9: Analysis outline.

division or modulo by 0. The `SafeMath` library is special in that it is not deployed as an own contract on the blockchain, but its functions get inlined during the compilation of a contract that uses them[5]. This specific behavior makes it particularly interesting to analyze the individual library functions as their concrete implementations may vary with changes in the compiler.

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

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

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

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

5.3 Large-scale Evaluation

We performed a series of experiments to assess the overall performance of our tool. In particular, we systematically evaluated *eThor*'s correctness and precision on the official EVM testsuite and additionally conducted a large scale analysis for the single-entrancy property, comparing *eThor* with the ZEUS [32] static analyzer, using the real-world data set introduced with the latter[6].

Automated Testing. For a principled correctness assessment, we evaluated *eThor* against the virtual machine test cases provided by the Ethereum Foundation[7]. Being formulated as pre- and postconditions, these test cases fall in the class of properties characterized in § 2.3 and we could automatically translate them into queries in *HoRSt*. The test suite defines 609 test cases, 604 of which specify properties relevant for a single contract setting (see § C.3 for details). Using a 1 second timeout, we were able to solve 85% (513) of the test cases precisely with a termination rate of 99% (597).

Reentrancy. For the call unreachability property described in Definition 3.2, we evaluated *eThor* against the set of real-world contracts presented in [32]. The authors extracted 22493 contracts from the Ethereum blockchain over a period of three months and (after deduplication) made available a list of 1524 contract addresses. Due to various problems of this data set (as described in [6]), sanitization leaves us with 720 distinct bytecodes out of which we label 100 contracts to be trivially non-reentrant (due to absence of possibly reentering instructions) and 2 were out of the analysis scope (containing at least one DELEGATECALL or CALLCODE instruction) and hence immediately classified to be potentially vulnerable. We make the sanitized benchmark available to the community, including bytecode and sources (where available) [6]. For 13 contracts we failed to reconstruct the control-flow graph, leaving us with 605 distinct contracts to run our experiments on.

We ran three different experiments for evaluating *eThor*'s performance for the single-entrancy property, performing no folding, linear folding, and exhaustive Horn clause folding. This experimental set up aims not only to conduct a comparison with ZEUS, but also to showcase how *eThor*'s modular structure facilitates its performance in that *eThor* can flexibly benefit from different optimization techniques of the *HoRSt* compiler. In the comparison with ZEUS, we take into account the combined result of the three different experiments (the contracts solvable using any of the applied foldings). For the exhaustive folding, we omitted instances where the time for `smt-lib` generation exceeded 15 minutes.[8] All of the experiments were conducted on a virtual machine with 64 Cores at 2 GHz and 128 GiB of RAM. At most 64 queries were executed at once, each with a 10 minutes timeout. Combining the different experiments we obtained results for all but 28 contracts.

We compared the results with [32]. Because of the existing soundness concerns regarding [32] in the literature [22, 45], we manually reassessed the ground truth for all contracts that were labeled insecure by at least one of the tools. Since this is a challenging and time consuming task, especially in the case that no Solidity source code is available, we excluded all contracts with more than 6000 bytecodes for which we were not able to obtain the source code, which leaves us with 709 contracts for which we assessed the ground truth.

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

[6] We compare with [32] as we found it the only (claimed) sound tool to support a property comparable to single-entrancy. [46] only supports a pattern (NW) which the authors claim to be different from reentrancy. [35] utilizes a similar pattern.

[7] https://github.com/ethereum/tests/

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

Measure	Definition	*eThor*	[32]
termination	*terminated*/*total*	94.3	98.1
sensitivity	$tp/(tp+fn)$	100	11.4
specificity	$tn/(tn+fp)$	80.4	99.8
F-measure	$2*(spec*sens/(spec+sens))$	89.1	20.4

Table 1: Performance comparison of *eThor* and ZEUS [32]. *total*/*terminated* denotes the number of contracts in the data set/the number of contracts the respective tool terminated on. *tp*/*fp* denotes the number of true/false positives and *tn*/*fn* the true/false negatives.

Surprisingly, we found numerous contracts labeled non-reentrant by [32] which, if analyzed in a single contract setting, definitely were reentrant according to the definition of reentrancy given in Definition 2.1 and also according to the informal definition provided in [32] itself[9]. We assume this to be an artifact of [32]'s syntactical treatment of the `call` directive on the Solidity level which is, however, insufficient to catch all possible reentrancies. As the work excludes reentrancies introduced by the `send` directive (even though this is officially considered potentially insecure [20]), for the sake of better comparability, we slightly updated our abstract semantics to account for calls that can be deemed secure following the same argument (namely that a small gas budget prevents reentrancy). We compare *eThor* against [32] on our manually established ground truth. The results are summarized in Table 1.

For achieving a termination rate comparable to [32] (94.3% vs. 98.1%), we needed to run our tool with a higher timeout (10 min. query timeout vs 1 min. contract time out for ZEUS). This difference can be explained by the fact that our analysis works on little structured bytecode in contrast to the simplified high-level representation used by [32]. Additional overhead needs to be attributed to the usage of sound abstractions on the bytecode level as well as to our different experimental setup that did not allow for the same amount of parallelization. The soundness claim of [32] is challenged by the experimentally assessed sensitivity of only 11.4%. One possible explanation for this low value, which deviates from the numbers reported in [32] on the same data set, is that the intuition guiding the manual investigation performed by [32] departed from the notion of single-entrancy and the intuitive definition given by the authors. This highlights the importance of formalizing not only the analysis technique but also the security properties to be verified. When interpreting the high specificity of [32] (almost 100%) one should consider that ZEUS labels only 25 contracts vulnerable in total out of which one is a false positive. Given that the data set is biased towards safe contracts (513 safe as opposed to 196 unsafe ones) a high specificity can be the result of a tool's tendency to label contracts erroneously secure. Due to the proven soundness, for *eThor* such a behavior is excluded by design. This overall advantage of *eThor* over ZEUS in terms of accuracy is reflected by *eThor*'s F-measure of 89.1% as opposed to 20.4% for ZEUS.

Horn Clause Folding. Our experimental evaluation shows that, while both forms of Horn clause folding improve the termination rate, the results of the different foldings are not directly comparable.

[9] [32] gives the following informal definition: 'A function is reentrant if it can be interrupted while in the midst of its execution, and safely re-invoked even before its previous invocations complete execution.'

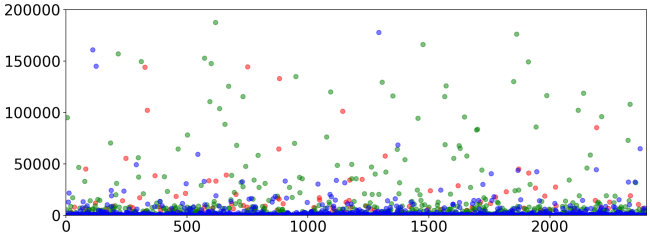

Figure 10: Query runtimes in ms for the combined approach itemized by queries. A red/green/blue dot denotes a query solved fastest with no/linear/exhaustive folding.

This is illustrated by Figure 10 which plots the (lowest) termination times for those queries that terminated within 200 seconds during the large-scale experiment. The different colors indicate the kind of optimization (no/linear/exhaustive folding) that was fastest to solve the corresponding query. The distribution of the dots shows that in the range of low query times (indicating structured contracts) exhaustive folding (depicted in blue) dominates. However, for longer query times, the linear folding (depicted in green) often shows a better performance. One possible explanation is that for more complex contracts, the blow-up in rules created by the exhaustive folding exceeds the benefits of eliminating more predicates. Interestingly, for few instances, even applying no folding at all (indicated in red) led to the fastest termination. We can only explain this behavior by special heuristics used inside *z3* that helped these particular cases. This shows the lacking predictability of *z3* and thereby motivates the necessity of high-level tools like *HoRSt* that allow users to easily combine different optimizations in order to obtain results reliably.

6 RELATED WORK & DISCUSSION

In the last years there have been plenty of works on the automatic analysis of Ethereum smart contracts. These works can be classified in dynamic and static analysis approaches: An example of a dynamic approach is the work by Grossmann et al. [24] which studies effectively callback freedom, a property characterizing the absence of reentrancy bugs, and provides a dynamic detection algorithm for it. Besides that, the authors prove that statically showing a smart contract to be effectively callback free is indeed undecidable. This work serves as a motivation why the correct and precise static analysis of smart contracts with respect to relevant security properties is challenging and requires the usage of suitable (sound) abstractions in order to be feasible, or even possible.

As a consequence, most practical static analysis tools so far focused on the heuristic detection of certain (classes of) bugs. These works do not strive for any theoretical guarantees nor do they aim for giving formal (and semantic) characterizations of security properties that the analysis targets. Important representatives of such bug-finding tools are the static analyzer Oyente [36] which was the first static analyzer for Ethereum smart contracts and the tool Osiris [45] which focuses on the detection of integer bugs. Since the aim and scope of these tools differ substantially from the ones of *eThor*, we omit a more detailed discussion and refer the interested reader to recent surveys [17, 22, 34] for more details.

In contrast to pure bug-finding, some recent works target a sound and automated static analysis of smart contract security properties. In particular the tools Securify [46], ZEUS [32], EtherTrust [22], and NeuCheck [35] make such soundness claims.

Securify implements a dependency analysis based on the reconstructed control-flow graph of contract bytecode and expresses some generic security properties in terms of these dependencies. The paper claims that the provided dependency patterns are either sufficient for the satisfaction (compliance patterns) or the violation (violation patterns) of a property. However, no proofs for the correctness of the control-flow graph transformation, the soundness of the dependency analysis itself, or the relationship between the security patterns and the properties are provided. As a consequence we could empirically show that Securify's algorithm for control flow reconstruction is unsound and give counterexamples for 13 out of 17 patterns (which partly also indicate a flaw in the dependency analysis itself). In [43] we give a detailed account of these issues.

NeuCheck is a tool for analyzing Solidity smart contracts by searching for patterns in the contract syntax graph. The soundness claim of the work is neither substantiated by a soundness statement, nor proof. Also no semantics of Solidity is given and no (formal) security properties are formulated. This lack of formalism makes it hard to validate any soundness claim. Despite the missing formal connections, the given patterns are clearly of syntactic nature and can be argued not to match the intuitive properties given throughout the paper which makes NeuCheck rather a bug-finding and style-checking tool. For more details we refer to [43].

Similar to NeuCheck, ZEUS analyzes smart contracts written in Solidity. To this end, it transforms Solidity smart contracts into an abstract intermediate language and later into LLVM bitcode which allows for leveraging existing symbolic model checkers. The code transformations are claimed to be semantics preserving which however has already been refuted by [22]. Additionally, the analyzed security properties are neither formally defined nor are they translated for model checking in a streamlined fashion: while some of them are compiled to assertions, other properties require additional code transformations which we show to be flawed in [43]. Empirical evidence for the unsoundness of ZEUS has been reported by [45] and is emphasized by the empirical evaluation in § 5.

The work presented in [22] surveys different approaches to static analysis and aims at illustrating design choices and challenges in sound static analysis. The work also discusses EtherTrust, a first proof of concept for a reachability analysis based on Horn clauses, which however is still preliminary and exhibits soundness issues in its abstraction as discussed [6].

For avoiding the pitfalls leading to unsoundness in the presented works, *eThor* follows a principled design approach: Starting from the formal EVM semantics defined in [23], it formulates an abstract semantics in the specification language *HoRSt* which is proven sound with respect to the concrete semantics, hence covering all particularities of the EVM bytecode language. Based on this abstract semantic specification, a streamlined compilation process creates an SMT encoding which is again systematically tested for soundness against the official test suite to minimize the effect of implementation bugs. The challenge of sound control flow reconstruction is solved by basing a corresponding preanalysis on a proper relaxation of the provably sound abstract semantics in the *Soufflé* format, ensuring that the original soundness guarantees are inherited. For a more robust development, it is planned to also streamline this process in the future by making the *HoRSt* compiler support *Soufflé* as an additional output format for a restricted Horn clause fragment. For providing end-to-end guarantees of the resulting static analyzer, we do not only ensure the soundness of the core analysis by proofs and testing, but also give provably sound approximations for relevant formalized semantic security properties suitable for encoding in the analysis framework.

7 CONCLUSION

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

This work opens up several interesting research directions. For instance, we plan to extend our analysis as well as *HoRSt* to relational properties, since some interesting security properties for smart contracts can be defined in terms of 2-safety properties [23]. Furthermore, we intend to further refine the analysis in order to enhance its precision, e.g., by extending the approach to a multi-contract setting, introducing abstractions for calls that approximate the account's persistent storage and local memory after calling more accurately. Furthermore, we plan to extend the scope of *HoRSt* significantly. First, we intend to make the specification of the static analysis accessible to proof assistants in order to mechanize soundness proofs. Furthermore, we intend to explore the automated generation of static analysis patterns from the specification of the concrete semantics, in order to further reduce the domain knowledge required in the design of static analyzers.

ACKNOWLEDGMENTS

This work has been partially supported by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement 771527-BROWSEC); by the Austrian Science Fund (FWF) through the projects PROFET (grant agreement P31621) and the project W1255-N23; by the Austrian Research Promotion Agency (FFG) through the Bridge-1 project PR4DLT (grant agreement 13808694) and the COMET K1 SBA; by the Internet Foundation Austria (IPA) through the netidee project EtherTrust (Call 12, project 2158); by the Vienna Business Agency through the project Vienna Cybersecurity and Privacy Research Center (VISP); and by Google Cloud.

REFERENCES

[1] 2016. The DAO Smart Contract. Available at http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code.

[2] 2017. *The Parity Wallet Breach, 30 million ether reported stolen.* Available at https://www.coindesk.com/30-million-ether-reported-stolen-parity-wallet-breach/.

[3] 2017. The Parity Wallet Vulnerability. Available at https://paritytech.io/blog/security-alert.html.

[4] 2019. SafeMath library source. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol.

[5] 2019. Solidity. https://solidity.readthedocs.io/.

[6] 2020. eThor: extended version, source code, build, and evaluation artifacts. https://secpriv.wien/ethor.

[7] 2020. SMT-LIB. Available at http://smtlib.cs.uiowa.edu/language.shtml.

[8] Chandra Adhikari. 2017. Secure Framework for Healthcare Data Management Using Ethereum-based Blockchain Technology. (2017).

[9] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. *CPP. ACM. To appear* (2018).

[10] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *International Conference on Principles of Security and Trust*. Springer, 164–186.

[11] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. 2016. Medrec: Using Blockchain for Medical Data Access and Permission Management. In *Open and Big Data (OBD), International Conference on*. IEEE, 25–30.

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

[13] Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. 2017. Findel: Secure Derivative Contracts for Ethereum. In *International Conference on Financial Cryptography and Data Security*. Springer, 453–467.

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

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

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

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

[18] Changyu Dong, Yilei Wang, Amjad Aldweesh, Patrick McCorry, and Aad van Moorsel. 2017. Betrayal, Distrust, and Rationality: Smart Counter-Collusion Contracts for Verifiable Cloud Computing. (2017).

[19] Ethereum 2018. *Solidity Docs.* Ethereum. https://solidity.readthedocs.io/en/develop/control-structures.html#error-handling-assert-require-revert-and-exceptions

[20] Ethereum 2019. *Solidity Docs.* Ethereum. https://solidity.readthedocs.io/en/v0.5.13/security-considerations

[21] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-gas Conditions in Ethereum Smart Contracts. *Object-Oriented Programming, Systems, Languages & Applications* OOPSLA (2018).

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

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

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

[25] Adam Hahn, Rajveer Singh, Chen-Ching Liu, and Sijie Chen. 2017. Smart Contract-Based Campus Demonstration of Decentralized Transactive Energy Auctions. In *Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), 2017 IEEE*. IEEE, 1–5.

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

[27] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. 2017. *KEVM: A Complete Semantics of the Ethereum Virtual Machine.* Technical Report.

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

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

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

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

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

[33] Johannes Krupp and Christian Rossow. 2018. TEETHER: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, 1317–1333.

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

[35] Ning Lu, Bin Wang, Yongxin Zhang, Wenbo Shi, and Christian Esposito. 2019. NeuCheck: A more practical Ethereum smart contract security analysis tool. *Software: Practice and Experience* (2019).

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

[37] Florian Mathieu and Ryno Mathee. 2017. Blocktix: Decentralized Event Hosting and Ticket Distribution Network. (2017). Available at https://blocktix.io/public/doc/blocktix-wp-draft.pdf.

[38] Patrick McCorry, Siamak F. Shahandashti, and Feng Hao. 2017. A Smart Contract for Boardroom Voting with Maximum Voter Privacy. *Proceedings of the Financial Cryptography and Data Security Conference* (2017).

[39] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. Available at http://bitcoin.org/bitcoin.pdf.

[40] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. *arXiv preprint arXiv:1802.06038* (2018).

[41] Benedikt Notheisen, Magnus Gödde, and Christof Weinhardt. 2017. Trading Stocks on Blocks-Engineering Decentralized Markets. In *International Conference on Design Science Research in Information Systems*. Springer, 474–478.

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

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

[44] Hisao Tamaki. 1984. Unfold/fold transformation of logic programs. *Proc. of 2nd ILPC* (1984), 127–138.

[45] Christof Ferreira Torres, Julian Schütte, et al. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference (SAC)*. ACM, 664–676.

[46] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS)*. ACM, 67–82.

[47] Zheng Yang and Hang Lei. 2019. FEther: An Extensible Definitional Interpreter for Smart-Contract Verifications in Coq. *IEEE Access* 7 (2019), 37770–37791.

[48] Ence Zhou, Song Hua, Bingfeng Pi, Jun Sun, Yashihide Nomura, Kazuhiro Yamashita, and Hidetoshi Kurihara. 2018. Security Assurance for Smart Contract. In *New Technologies, Mobility and Security (NTMS), 2018 9th IFIP International Conference on*. IEEE, 1–5.

APPENDIX

The appendix is structured as follows: In § A we overview the analysis specification language *HoRSt*. In § B we make the theoretical foundations of our work explicit, in particular its relation to abstract interpretation. Finally, § C gives details on how the security properties discussed in the paper are implemented in *eThor* using the specification language *HoRSt*.

A *HORST*

This section gives an introduction to the newly developed language *HoRSt* that allows for the high-level specification of Horn-clause-based static analyses. We will first give a short primer that illustrates the main functionality of *HoRSt*, followed by a more detailed discussion of the optimizations performed by the *HoRSt* compiler.

A.1 *HoRSt* by Example

For illustrating the features of *HoRSt* we show how to express a general rule for binary stack operations, subsuming the rule for addition presented in § 3. Figure 11 shows an excerpt of the *HoRSt*-specification of the presented static analysis. The abstract domain of the analysis is realized by the definition of the abstract data type `AbsDom`. Predicate signatures can be specified by corresponding predicate declarations as done for the case of the `MState` predicate. *HoRSt* allows for parametrizing predicates and thereby specifying whole predicate families: The `MState` predicate is parametrized by two integer values (as specified in the curly braces) that will intuitively correspond to the contract's identifier and the program counter whose state it is approximating. The arguments of the `MState` predicate family reflect exactly those specified in § 3.

To facilitate modular specifications, *HoRSt* supports non-recursive operations over arbitrary types, such as `absadd` which implements abstract addition. In the given example, we show the flexibility of *HoRSt* by presenting a single rule template for generating rules for all binary stack operations. To this end, we define a function `binOp` that given an opcode `c` and two integer arguments applies to them the binary operation corresponding to the opcode. This function is then leveraged in the rule template `opBin`. Rule templates serve for generating the abstract semantics given in the form of Horn clauses. As in our case the abstract semantics is specified as a function on a concrete contract, the generation of Horn clauses in *HoRSt* needs to be linked to a concrete contract bytecode. In order to account for that in a generic fashion given that *HoRSt* cannot support facilities for reading files or parsing bytecodes, *HoRSt* provides an interface for interacting with custom relations generated by *Java*™ code. This interface is specified upfront by so called *selector functions* (introduced with the key word `sel`) which are declared, but not defined in the *HoRSt* specification. In the given example, we declare selector functions for accessing the identifiers of the contracts to be analyzed (`ids`), the set of binary operations (`binOps`) and the program counters in a contract that hold opcodes of a specific type (`pcForIdAndOpcode`). In addition to that, selector functions also allow for more advanced functionalities such as incorporating the results of a preanalysis in an elegant fashion: To this end, we declare the selector function `argumentsTwoForIdAndPc` that returns arguments to the operation that could be statically precomputed (returning −1 in case of failure). For generating Horn clauses, we can parametrize the rule over the cross product of the result of (nested) selector function applications as done in for the `opBin` rule. This then exactly generates Horn clauses abstracting the behavior of a binary stack operation as discussed in § 3: A stack size check is performed, the two arguments are selected from the stack and finally the `MState` predicate at the next program counter is implied with an updated stack having the operation's result as top element. The only derivation occurs due to the consideration of the preanalysis: the operation `tryConcrete` tries to access the statically precomputed argument, and only in case of its absence performs the (more expensive) stack access. This step however is not a necessity, but just illustrates how the interplay between different stages of a static analysis can be implemented for boosting the performance.

A.2 Compiler Optimizations

The *HoRSt* compiler features several transformations to generate optimized `smt-lib` output. It first resolves high-level language constructs such as operations and data types, and unfolds the parametrization introduced by selector functions. The resulting basic Horn clause representation is optimized by constant folding, and (optionally) performing different flavors of the unfolding transformation to eliminate predicates that are not relevant for the queries. In the following, we explain the unfolding transformation in more detail.

The idea behind the unfolding transformation is that a predicate p can be eliminated from a set of Horn clauses Λ by unfolding the occurrences of p in the premises according to the clauses that have p as conclusion. An example is given in Figure 12. Here predicate P_2 is eliminated by merging the two single execution steps (modeled by the two clauses on the left) into a combined clause (on the right) summarizing the steps.

This intuition serves as a starting point for the unfolding strategy of *linear folding*. In linear folding, all clauses representing a basic block of sequential execution steps are merged into a single clause. More precisely, the unfolding transformation is only applied to those predicates that are used linearly in Λ, meaning that p occurs in the premises of exactly one clause in Λ and in the conclusion of exactly one different clause in Λ. Linear folding has the advantage that it runs linearly in the number of clauses in Λ and yields as result a reduced set of clauses Λ' such that $|\Lambda'| \leq |\Lambda|$.

In contrast, applying the unfolding transformation exhaustively on all predicates (with exception of those that are recursively used) might yield an exponential blow-up in clauses (and hence also result in exponential runtime). In practice however, the set of clauses Λ' resulting from such a *exhaustive folding* is often of a reasonable size. For mitigating the runtime overhead, however, it is crucial to avoid unnecessary blow-ups in the intermediate clause sets produced during the transformation: To this end, for exhaustive folding *HoRSt* applies linear folding first and only afterwards performs the unfoldings that multiply existing clauses.

Figure 13 shows the effects of linear and exhaustive folding for a simple contract with a loop. Since the abstraction at each program counter is modeled by an own predicate (MState, here M for short), the contract control flow is reflected by the logical dependencies between these predicates as defined in the Horn clauses of the abstract semantics. Hence, we depict the abstract semantics as a transition system, interpreting predicates as states and Horn clauses as transitions. Linear folding collapses all sequences with linear control flow, while exhaustive folding in this case reduces the state space even further without adding additional clauses.

```
datatype AbsDom := @T | @V<int>; // Abstract Domain
datatype Opcode := @STOP | @ADD | ... | @INVALID | @SELFDESTRUCT // opcodes (shortened)

pred MState{int*int}: int * array<AbsDom> * array<AbsDom> * array<AbsDom> * bool;

op absadd(a: AbsDom, b: AbsDom): AbsDom := match (a, b) with | (@V(x), @V(y)) => @V((x + y) mod MAX) | _ => @T;
op binOp(c: Opcode, x: AbsDom, y: AbsDom): AbsDom := match c with | @ADD => absadd(x, y) | ... | _ => @T;

sel ids: unit -> [int]; // contracts to be analyzed
sel binOps: unit -> [int]; // binary stack operations
sel pcsForIdAndOpcode: int * int -> [int]; // program counters at which a specific opcode occurs in a specific contract
sel argumentsTwoForIdAndPc: int * int -> [int * int]; // results from the preanalysis for a given contract and pc

op tryConcrete{!c:int}(val:AbsDom): AbsDom := (!c = ~1) ? (val) : (@V(!c));

rule opBin :=
  for (!op: int) in binOps(), (!id: int) in ids(), (!pc: int) in pcsForIdAndOpcode(!id, !op),
      (!a:int, !b: int) in argumentsTwoForIdAndPc(!id, !pc)
    clause [?x: AbsDom, ?y:AbsDom, ?size: int, ?sa: array<AbsDom>, ?mem: array<AbsDom>, ?stor: array<AbsDom>, ?cl: bool]
      MState{!id, !pc}(?size, ?sa, ?mem, ?stor, ?cl), ?size > 1,
      ?x = tryConcrete{!a} (select ?sa (?size -1)), ?y = tryConcrete{!b} (select ?sa (?size -2))
      => MState{!id, !pc +1}(?size -1, store ?sa (?size -2) (binOp(intToOpCode(!op), ?x,?y)), ?mem, ?stor, ?cl);
```

Figure 11: *HoRSt* rule describing the abstract semantics of local binary stack operations.

$$\left.\begin{array}{l} P_1(x) \wedge y = x+1 \Rightarrow P_2(y) \\ P_2(y) \wedge z = y*3 \Rightarrow P_3(z) \end{array}\right\} P_1(x) \wedge y = x+1 \wedge z = y*3 \Rightarrow P_3(z)$$

Figure 12: Unfolding of P_2.

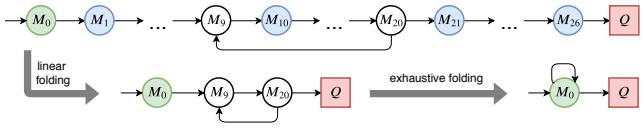

Figure 13: Example of linear and exhaustive folding. Transition system view of the abstract semantics: States denote predicates and arrows denote Horn clauses having the start predicate as premise and the goal predicate as conclusion. Initial (final) states are colored green (red). Linearly used predicates are colored blue.

B THEORETICAL FOUNDATIONS OF *ETHOR*

In this section, we provide details on the theoretical foundations of *eThor*. We first formally characterize the notion of Horn-clause-based abstractions as they can be implemented in *HoRSt* and then relate this concept to the framework of abstract interpretation.

B.1 Horn-clause-based Abstractions

In this section, we more formally characterize the aim and scope of this work, as well as the kind of static analyses that are realizable by *HoRSt*. Generally, we focus on the reachability analysis of programs with a small-step semantics, which we over-approximate by an abstract program semantics based on Horn clauses. More formally, we will assume a program's small-step semantics to be a binary relation S_s over program configurations $c \in C$. A Horn-clause-based abstraction for such a small-step semantics S_s is then fully specified by a tuple $(\mathcal{D}, \mathcal{S}, \alpha, \Lambda)$ where $\mathcal{S}$ defines the signature of predicates with arguments ranging over (partially) ordered subsets of $\mathcal{D}$. For a given predicate signature $\mathcal{S}$, an abstraction function $\alpha : C \to \mathcal{A}$ maps concrete program configurations $c \in C$ to abstract program configurations $\Delta \in \mathcal{A}$ consisting of instances of predicates in $\mathcal{S}$.

Formally, a predicate signature $\mathcal{S} \in \mathcal{N} \twoheadrightarrow \prod(\mathcal{P}(\mathcal{D}) \times (\mathcal{P}(\mathcal{D}) \times \mathcal{P}(\mathcal{D})))$ is a partial function from predicate names $\mathcal{N}$ to their argument types (formally written as a product over the subsets of some abstract superdomain $\mathcal{D}$, equipped with a corresponding order). We require for all $n \in \mathcal{N}$ that $(D, \leq) \in \mathcal{S}(n)$ such that $(D, \leq)$ forms a partially ordered set. Correspondingly, the set of abstract configurations $\mathcal{A}_\mathcal{S}$ over $\mathcal{S}$ can be defined as $\mathcal{P}(\{n(\vec{v}) \mid n \in \mathcal{N} \wedge \forall i \in \{1, \ldots, |\mathcal{S}(n)|\}. \pi_i(\mathcal{S}(n)) = (D, \leq) \implies \pi_i(\vec{v}) \in D\})$ where $\pi_i(\cdot)$ denotes the usual projection operator. The abstraction of a small-step semantics is then a set of constrained Horn clauses $\Lambda \subseteq \mathcal{H}(\mathcal{S})$ that approximates the small-step execution rules.

A constrained Horn clause is a first order formula of the form

$$\forall X.\ \Phi, P \Rightarrow c$$

Where $X \subseteq Vars \times \mathcal{P}(\mathcal{D})$ is a (functional) set of typed variables, and Φ is a set of quantifier-free constraints over the variables in X. Conclusions c are predicate applications $n(\vec{z}) \in P_X := \{n(\vec{x}) \mid |\vec{x}| = |\mathcal{S}(n)| \wedge \forall i \in \{1, \ldots, |\vec{x}|\}. \pi_i(\vec{x}) = x \wedge \pi_i(\mathcal{S}(n)) = (D, \leq) \implies (x, D) \in X\}$ over variables in X that respect the variable type. Correspondingly, the premises $P \subseteq P_X$, are a set of predicate applications over variables in X.

We lift the suborders of $\mathcal{S}$ to an order on abstract configurations $\Delta_1, \Delta_2 \in \mathcal{A}_\mathcal{S}$ as follows:

$$n_1(\vec{t_1}) \leq_p n_2(\vec{t_2}) := n_1 = n_2 \wedge \forall i \in \{1, \ldots, |\vec{t_1}|\}. \pi_i(\vec{t_1}) \leq_{n_1,i} \pi_i(\vec{t_2})$$

given $\pi_i(\mathcal{S}(n)) = (D_{n,i}, \leq_{n,i})$

$$\Delta_1 \leq \Delta_2 := \forall p_1 \in \Delta_1.\ \exists p_2 \in \Delta_2.\ p_1 \leq_p p_2$$

Finally, we introduce the notion of soundness for a Horn-clause-based abstraction.

Definition B.1. A Horn-clause-based abstraction $(\mathcal{D}, \mathcal{S}, \alpha, \Lambda)$ *soundly approximates* a small-step semantics S_s if

$$\forall (c, c') \in S_s^*.\ \forall \Delta.\ \alpha(c) \leq \Delta \Rightarrow \exists \Delta'.\ \Delta, \Lambda \vdash \Delta' \wedge \alpha(c') \leq \Delta' \quad (1)$$

This statement requires that, whenever a concrete configuration c' is reachable from configuration c (meaning that (c, c') is contained in the reflexive and transitive closure of S_s, denoted as S_s^*), it shall hold that from all abstractions Δ of c, the Horn clause abstraction allows us to logically derive ($\vdash$) a valid abstraction Δ' of c'. Note that α intuitively yields the most concrete abstraction of a configuration, hence to make the property hold for all possible abstractions of a configuration, we strengthen the property to hold for all abstractions that are more abstract than $\alpha(c)$. The soundness theorem implies that whenever we can show that from some abstraction Δ of a configuration c there is no abstract configuration Δ' derivable such that Δ' abstracts c', then c' is not reachable from c. Consequently, if it is possible to enumerate all abstractions of c', checking non-derivability (as it is supported by the fixedpoint engines of modern SMT solvers) gives us a procedure for proving unreachability of program configurations.

B.2 Relation to Abstract Interpretation

It is possible to phrase the previous characterization in terms of classical abstract interpretation notions. More precisely, we can define a Galois connection (α, γ) between sets of concrete configurations $\mathcal{P}(C)$ (ordered by $\subseteq$) and abstract configurations $\mathcal{A}$ (ordered by $\leq$). To this end, we lift the abstraction function α to sets of configurations in a canonical fashion:

$$\alpha(C) := \bigcup_{c \in C} \alpha(c) \tag{2}$$

Next, we define the concretization function based on α:

$$\gamma(\Delta) := \{c \in \mathcal{C} \mid \alpha(c) \leq \Delta\}$$

Lemma B.2. *The pair of functions (α, γ) forms a Galois connection between $(\mathcal{P}(C), \subseteq)$ and $(\mathcal{A}, \leq)$.*

Proof. We need to show for all C and Δ that

$$\alpha(C) \leq \Delta \Leftrightarrow C \subseteq \gamma(\Delta)$$

$\Rightarrow$: Let $\alpha(C) \leq \Delta$. Further let $c \in C$. We show that $c \in \gamma(\Delta)$. By the definition of γ it is sufficient to show that $\alpha(c) \leq \Delta$. Let $p_1 \in \alpha(c)$. We show that there is some $p_2 \in \Delta$ such that $p_1 \leq p_2$. Since $p_1 \in \alpha(c)$ and $c \in C$, we know that $p_1 \in \alpha(C)$ and since $\alpha(C) \leq \Delta$ also that there needs to be some $p_2 \in \Delta$ such that $p_1 \leq p_2$ what concludes the proof.

$\Leftarrow$: Let $C \subseteq \gamma(\Delta)$. Further let $p_1 \in \alpha(C)$. We show that there is some $p_2 \in \Delta$ such that $p_1 \leq p_2$. Since $p1 \in \alpha(C)$ there must be some $c \in C$ such that $p_1 \in \alpha(c)$. And from $C \subseteq \gamma(\Delta)$ we can conclude that $c \in \gamma(\Delta)$ which implies that $\alpha(c) \leq \Delta$. Consequently there needs to be a $p_2 \in \Delta$ such that $p_1 \leq p_2$ what concludes the proof.

□

Now, we can define reachability of concrete configurations and derivability of abstract configurations as the least fixed points of step functions (F_I for concrete configuration steps and F'_{Δ_I} for abstract configuration steps) which describe a collecting semantics (with respect to some initial configuration).

$$F_I(C) := \{c' \mid \exists c \in C.\, (c, c') \in S_s\} \cup I$$

$$F'_{\Delta_I}(\Delta) := \{p \mid \Delta, \Lambda \vdash p\} \cup \Delta_I$$

We obtain the following intuitive correspondences between the different characterizations:

$$(c, c') \in S_s^* \Leftrightarrow c' \in \mathit{lfp}[F_{\{c\}}] \tag{3}$$

$$\Delta, \Lambda \vdash \Delta' \Leftrightarrow \Delta' \subseteq \mathit{lfp}[F'_{\Delta}] \tag{4}$$

where $\mathit{lfp}[f]$ denotes the least fixed point of a function f.

To ensure that the corresponding least fixed points exists, we need to ensure that the domains $\mathcal{P}(C)$ and $\mathcal{A}$ of the Galois connection form a complete lattice and that both F_I and F'_{Δ_I} are monotone. While $\langle \mathcal{P}(C), \subseteq, \emptyset, \mathcal{P}(C), \cup, \cap \rangle$ is the canonical power set lattice, we can easily show $\langle \mathcal{A}, \leq, \emptyset, \Delta, \cup, \cap \rangle$ to also form a complete lattice as $\subseteq$ is a subrelation of $\leq$. While it is trivial to show that F_I is monotone, for F'_{Δ_I} it becomes a proof obligation on Λ:

$$\forall \Delta, \Delta'.\, \Delta \leq \Delta' \wedge \Delta, \Lambda \vdash p \implies \exists p'.\, p \leq p' \wedge \Delta', \Lambda \vdash p' \tag{5}$$

Using the step functions, we can characterize sound over-approximations as defined in Definition B.1 in an alternative fashion. More precisely, we require our approximation to be a *sound upper approximation* [16].

Lemma B.3. *A Horn-clause-based abstraction $(\mathcal{D}, \{\leq_{n,i}\}_{(n,i)}, \mathcal{S}, \alpha, \Lambda)$ soundly approximates a small-step semantics S_s iff Λ satisfies Equation (5) and for all $c \in C$ and all $\Delta \geq \alpha(c)$*

$$\alpha(\mathit{lfp}[F_{\{c\}}]) \leq \mathit{lfp}[F'_{\Delta}]$$

Proof. "$\Rightarrow$": Assume Equation (1) and $f_1 \in \alpha(\mathit{lfp}[F_{\{c\}}])$ for some fact f_1. We show that there exists some fact f_2 such that $f_2 \in \mathit{lfp}[F'_{\Delta}]$ and $f_1 \leq f_2$. By Equation (2), we know that from $f_1 \in \alpha(\mathit{lfp}[F_{\{c\}}])$ we can conclude that there exists some $c' \in \mathit{lfp}[F_{\{c\}}]$ such that $f_1 \in \alpha(c')$. By Equation (3), we have that $(c, c') \in S_s^*$ and hence by Equation (1) we can conclude that there exists some Δ' such that $\Delta, \Lambda \vdash \Delta'$ and $\alpha(c') \leq \Delta'$. With $f_1 \in \alpha(c')$ we get from this that there exists some $f_2 \in \Delta'$ such that $f_1 \leq f_2$. Since $\Delta, \Lambda \vdash \Delta'$, we get from Equation (4) that $\Delta' \subseteq \mathit{lfp}[F'_{\Delta}]$ and hence also $f_2 \in \mathit{lfp}[F'_{\Delta}]$ which concludes the proof.

"$\Leftarrow$": Assume $\alpha(\mathit{lfp}[F_{\{c\}}]) \leq \mathit{lfp}[F'_{\Delta}]$ and let $(c, c') \in S_s^*$ and $\alpha(c) \leq \Delta$. We show that there is some Δ' such that $\Delta, \Lambda \vdash \Delta'$ and $\alpha(c') \leq \Delta'$. By Equation (3), we get that $c' \in \mathit{lfp}[F_{\{c\}}]$ and hence also $\alpha(c') \subseteq \alpha(\mathit{lfp}[F_{\{c\}}])$ (by Equation (2)). As $\alpha(\mathit{lfp}[F_{\{c\}}]) \leq \mathit{lfp}[F'_{\Delta}]$ it follows that also $\alpha(c') \leq \mathit{lfp}[F'_{\Delta}]$. Additionally, it follows from Equation (4) that $\Delta, \Lambda \vdash \mathit{lfp}[F'_{\Delta}]$. This closes our proof. □

Given that F'_{Δ} is monotonic, $\alpha(\mathit{lfp}[F_{\{c\}}]) \leq \mathit{lfp}[F'_{\Delta}]$ can be shown to follow from the one-step characterization below:

$$\alpha \circ F \leq F' \circ \alpha \tag{6}$$

(where $F = F_{\emptyset}$ and $F' = F'_{\emptyset}$).

This is because $\alpha \circ F \leq F' \circ \alpha$ implies for all $c \in C$ and all $\Delta \geq \alpha(c)$ that $\alpha \circ F_{\{c\}} \leq F'_{\Delta} \circ \alpha$ and by the fixed point transfer theorem [16] for Galois connections, this result can be lifted to least fixed points. As a consequence for proving Theorem 3.1, it is sufficient to show that Equation (5) and Equation (6) hold.

B.3 Analysis Definition (continued)

We overview additional details of the analysis definition in § 3.

The orders on the abstract argument domains of the predicates in Figure 4 are formally defined as follows:

$$\begin{aligned}
\leq_{\hat{D}} &:= \{(\hat{a},\hat{b}) \mid \hat{b} = \top \vee \hat{a} = \hat{b}\} \\
\leq_{\mathbb{N}} &:= \{(m,n) \mid m = n\} \\
\leq_{\mathbb{B}} &:= \{(a,b) \mid a = b\} \\
\leq_{\mathbb{N}\to\hat{D}} &:= \{(f,g) \mid \forall n \in \mathbb{N}.\ f(n) \leq_{\hat{D}} g(n)\} \\
\leq_{\mathbb{N}\times(\mathbb{N}\to\hat{D})} &:= \{((m,f),(n,g)) \mid m = n \ \wedge\ \forall i < m.\ f(i) \leq_{\hat{D}} g(i)\}
\end{aligned}$$

We assume that the same orders apply to the same argument domains of different predicates.

Abstract operations. We formally define abstract operations on values from the abstract argument domains, starting with binary operations on natural numbers: Let $op_{bin} \in \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ be a binary operation. We define abstract binary operations as follows:

$$\widehat{\cdot} \in (\mathbb{N}\times\mathbb{N}\to\mathbb{N}) \to \hat{D}\times\hat{D}\to\hat{D}$$

$$\widehat{op_{bin}}\ (\hat{x},\hat{y}) := \begin{cases} op_{bin}(\hat{x},\hat{y}) & \hat{x},\hat{y} \in \mathbb{N} \\ \top & \text{otherwise} \end{cases}$$

Similarly, we can define abstract comparison operators. Let $op_{comp} \in \mathbb{N}\times\mathbb{N}\to\mathbb{B}$ be a comparison operation on natural numbers. We define abstract comparison operations as follows:

$$\widehat{\cdot} \in (\mathbb{N}\times\mathbb{N}\to\mathbb{B}) \to \hat{D}\times\hat{D}\to\mathbb{B}$$

$$\widehat{op_{comp}}\ (\hat{x},\hat{y}) := \begin{cases} op_{comp}(\hat{x},\hat{y}) & \hat{x},\hat{y} \in \mathbb{N} \\ 1 & \text{otherwise} \end{cases}$$

We further define the abstract operations for array access used in Figure 5. First, we define the function for extracting a specified fraction of an integer (interpreted as 32-byte word)

$$\cdot_{[\cdot,\cdot]} \in \hat{D}\times\mathbb{N}\times\mathbb{N}\to\hat{D}$$

$$\hat{v}_{[l,r]} := \begin{cases} \left\lfloor \frac{\hat{v}}{256^{31-r}} \right\rfloor \bmod 256^{r-l+1} & l \leq r \ \wedge\ \hat{v} \in \mathbb{N} \\ \top & \text{otherwise} \end{cases}$$

Next, we define the append function:

$$||_{\cdot} \in \hat{D}\times\hat{D}\times\mathbb{N}\to\hat{D}$$

$$\hat{v}||_n\hat{w} := \begin{cases} \hat{w} * 256^n + \hat{v} & \hat{v},\hat{w} \in \mathbb{N} \\ \top & \text{otherwise} \end{cases}$$

We focused here only on those operations that were used in § 3. For a full account of all abstract operations, we refer the reader to our *HoRSt* specification in [6].

C CHECKING SECURITY PROPERTIES WITH *ETHOR*

In this section, we discuss how the security properties presented in § 2.3 are implemented in *eThor* using *HoRSt*. In particular, we explain how reachability properties can be abstracted as queries using the example of the call reachability property. We then illustrate the infrastructure for proving functional correctness properties as well as the one for automated soundness and precision testing.

C.1 From Reachability Properties to Queries

All reachability properties introduced in § 3.5 can be seen as instances of properties of the following form:

$$\mathcal{R}(P,R) := \forall s.\ P([s]) \implies \neg\exists S',\ \Gamma \vdash s_{c^*}::S \to^* S' +\!\!+ S \ \wedge\ R(S')$$

where s is assumed to be strongly consistent with c^* and S' is assumed to be non-empty. We will refer to properties of this form in the following as *unreachability* properties.

For the sake of presentation, we will in the following interpret predicates P, R as the sets of elements satisfying these predicates. Additionally, we will overload the abstraction function α to operate on sets of configurations hence writing $\alpha_{c^*}(R)$ for $\bigcup_{S'\in R}\alpha_{c^*}(S')$ and $\alpha_{c^*}(P)$ for $\bigcup_{[s_{c^*}]\in P}\alpha_{c^*}([s_{c^*}])$.

Following Theorem 3.1 for proving such properties it is sufficient to give some set Δ_P such that $\Delta_P \geq \alpha_{c^*}(P)$ and to show, for any set Δ_R over-approximating $\alpha_{c^*}(R)$ that $\Delta_P \not\vdash \Delta_R$. Instead of showing this property for all possible sets Δ_R, it is sufficient to find a query set Δ_{query} that shares at least one element with all possible sets Δ_R:

$$\forall\Delta_R.\ \Delta_R \geq \alpha_{c^*}(R) \implies \Delta_R \cap \Delta_{query} \neq \emptyset \tag{7}$$

Proving $\Delta_R, \delta(c^*) \not\vdash \Delta_{query}$ then implies that $\mathcal{R}(P,R)$ holds.

Usually, such a set can be easily constructed from R as follows:

$$\Delta_{query}(R) := \{p' \mid \exists p.\ p \in \alpha_{c^*}(R) \ \wedge\ p \leq_p p'\} \tag{8}$$

Intuitively, it is sufficient to query for the most concrete abstraction (as given by α_c^*) of the concrete configurations in R and all predicate-wise ($\leq_p$) coarser abstractions of those. The set $\Delta_{query}(R)$ however is only a valid query set for R if for some $S' \in R$ it holds that $\alpha_{c^*}(S')$ is non-empty. Otherwise Equation (7) is trivially violated. Intuitively this means that only postconditions R that make some restrictions on those callstack components that are modeled by the analysis (e.g., executions of contract c^*) can be reasonably analyzed using this technique. We formally state this property in the following lemma:

Lemma C.1. *Let $S_s \subseteq C\times C$ be a small-step semantics and $(\mathcal{D},\mathcal{S},\alpha,\Lambda)$ a sound abstraction thereof. Further, let $P, R \subseteq C$ be predicates on configurations and Δ_P be an abstract configuration s.t. $\Delta_P \geq \alpha(P)$. Then if there is some $c' \in R$ s.t. $\alpha(c') \neq \emptyset$ it holds that*

$$\Delta_P, \Lambda \not\vdash \Delta_{query}(R) \implies \mathcal{R}(P,R)$$

Thus, it is generally sufficient to query for the reachability of $\Delta_{query}(R)$ in order to prove an unreachability property $\mathcal{R}(P,R)$.

We will next show how this theoretical result can be used in practice and in particular at the level of *HoRSt*.

Initialization. For checking an unreachability property $\mathcal{R}(P,R)$, we need to show the non-derivability of a valid query set Δ_{query} from some abstract configuration $\Delta_P \geq \alpha(P)$. This requires to axiomatize such an abstract configuration Δ_P. This can be easily done in *HoRSt* by providing rules having `true` as a single premise. For axiomatizing that the execution starts in an initial machine state as required for the call unreachability property defined in Definition 3.2 we can add the following rule to the analysis specification:

```
1 rule initOp :=
2  clause true => MState{0}(0, [@V(0)], [@V(0)], [@T], false);
```

As the precondition P of the call unreachability property requires the top state s (that also serves as the zero-bar for the call level) to be initial, $\alpha(s)$ can contain only predicate applications of the form

```
query reentrancyCall
  for (!id: int) in ids(),
      (!pc:int) in pcsForIdAndOpcode(!id, CALL)
  [?sa: array<AbsDom>, ?mem: array<AbsDom>,
     ?stor: array<AbsDom>, ?size:int]
    MState{!id, !pc}(?size, ?sa, ?mem, ?stor, true);
```

Figure 14: *HoRSt*-query for reentrancy.

$\mathsf{MState}_0((0, \lambda x.\,0), \lambda x.\,0, m, 0)$ where m is some memory mapping. However, $\lambda x.\,\top$ (`[@T]` in *HoRSt*) over-approximates all memory arrays and hence $\Delta_P = \{\mathsf{MState}_0((0, \lambda x.\,0), \lambda x.\,0, \lambda x.\,\top, 0)\} \geq \alpha(P)$.

Queries. In addition to syntax for writing an analysis specification, *HoRSt* also provides mechanisms for the interaction with the underlying SMT solver. More precisely it supports syntax for specifying *queries* and *tests*. Syntactically, queries consist of a list of premises (as in a clause). A query leads to the invocation of the SMT solver to test whether the conjunction of those premises is derivable from the given initialization using the specified rules. The query will result in `SAT` in case that all premises are derivable and in `UNSAT` in case that the conjunction of premises can be proven to be non-derivable.

In order to check for reachability of abstract configurations, *HoRSt* allows for the specification of (reachability) queries that can also be generated from selector functions. The query shown in Figure 14 for instance checks for reentrancy by checking if any CALL instruction is reachable at call level 1[10]. It therefore is an implementation of the reachability property introduced in § 3.5. This query can be obtained from the call unreachability property defined in Definition 3.2 which is of the form $\mathcal{R}(P, R)$ with $R := \{s_{c^*}::S' \mid |S'| > 0 \ \wedge \ c^*.code[s, \mu.\mathrm{pc}] \in Inst_{call}\}$. Intuitively, we can split this property into a set of different properties $\mathcal{R}(P, R_i)$ where i ranges over the set of CALL instructions in c^*. More precisely, let $R_i := \{s_{c^*}::S' \mid |S'| > 0 \ \wedge \ s, \mu.\mathrm{pc} = i\}$ then it holds that

$$\mathcal{R}(P, R) \Leftrightarrow \forall i \in \{i \mid c^*.code[i] \in Inst_{call}\}.\, \mathcal{R}(P, R_i)$$

Then each instance of the `reentrancyCall` query specifies one query set Δ^i_{query} that satisfies Equation (7) for R_i. Thus showing the underivability of all those sets from Δ_P proves the claim. Intuitively, Δ^i_{query} satisfies Equation (7) for R_i because $\alpha^*_c(R_i)$ contains an application of a predicate MState_i with argument $c\ell = 1$ and so it needs to contain all abstractions of $\Delta_{R_i} \geq \alpha^*_c(R_i)$ as the $c\ell$ component cannot further be abstracted. Consequently, the set Δ^i_{query}, which contains all predicates of that form, has a trivial intersection with Δ_{R_i}.

C.2 Functional Correctness

For checking functional correctness, some modifications to the abstract semantics are necessary. This is because the different contract executions need to be bound the corresponding input data of the call and to account for return data. We will in the following overview the relevant changes and motivate that similar modifications can easily be incorporated for reasoning about other dependencies with the execution or blockchain environment. We will present the relevant modifications in *HoRSt* syntax so that the explanations serve as a guide to the enhanced version of the semantics [6].

[10]There are corresponding queries for other relevant transaction initiating instructions.

First, the relevant predicates need to be enriched with a corresponding representation of the call data. We decided to represent call data as a word array with the particularity that the array's first element represents only 4 bytes. This is due to the call conventions enforced by the Solidity compiler which interpret the first 4 bytes of input data as the hash of the called function's signature to dispatch function calls properly. In addition to the call data, we introduce a new predicate representing the return data of a call.

Formally, we arrive at the following predicate definitions:

```
datatype CallData := @D<int*array<AbsDom>>;
pred MState{int*int}: int * array<AbsDom> * array<AbsDom> *
      array<AbsDom> * bool * CallData;
pred Exc{int}: bool;
pred Halt{int}: array<AbsDom> * AbsDom * bool * CallData;
pred ReturnData{int}: int * AbsDom * bool * CallData;
```

Note that we represent call data as a pair of its size and an array of abstract words. Also, we added to the `Halt` predicate an argument representing the return data size. This argument is from the abstract domain with `@T` indicating that the concrete size of the return data is unknown. The `ReturnData` predicate maps the positions of the return data (word) array to the corresponding values that it holds.

The existing rules simply propagate the call data array with the only addition that the CALLDATALOAD instruction now accesses the call data array instead of over-approximating the loaded value. The new rule for CALLDATALOAD is depicted in Figure 15. The CALLDATALOAD operation takes as argument a value from the stack that specifies the position in the byte array where a word should be accessed. The rule is split into two clauses to benefit from the preanalysis. In case that the position of call data is known upfront, the call data array `?call` can be accessed more precisely. Since we model the call data as a word instead of a byte array (similar to our memory abstraction), either a word loaded from it consists of a full word in the word array or needs to be composed out of two neighboring words. Composing two integers (interpreting them as byte arrays) however requires exponentiation as defined in the append function in § B.3. *z3* is not able to handle general exponentiation - for this reason we can only compute such exponentiations (by unfolding to multiplications) whose exponent is known upfront. Thus, the first rule in Figure 15 handles the case where the argument to the call is known upfront: the `accessWordCallData` function expects the position as a parameter and computes the accessed word precisely from the call data array by unrolling exponentiation. The second rule handles the case where the argument to the call is not known upfront. In case that during the analysis it can be detected to be concrete (by the function `isConcrete`), the `accessWordCalldataEven` function is used to access the call data at the corresponding position. This function however only yields a precise result in case that the provided position corresponds to the beginning of a word in the call data array, otherwise it over-approximates the result as `T`.

The `ReturnData` predicate is inhabited by the rules that model regular halting. We exemplarily show the rule of the RETURN opcode depicted in Figure 16. The RETURN instruction in EVM reads the memory offset and the return data length from the stack and returns the corresponding memory fragment as byte array. In our abstraction the return data is modeled by an own predicate that holds words instead of bytes. This design choice follows the one made for the word-indexed memory and the call data array which hold words instead of bytes as well for performance reasons. The

```
rule opCallDataLoad :=
for (!id: int) in ids(), (!pc: int) in pcsForIdAndOpcode(!id, CALLDATALOAD), (!a: int) in argumentsOneForIdAndPc(!id, !pc)
clause [?x: AbsDom, ?size: int, ?sa: array<AbsDom>, ?mem: array<AbsDom>, ?stor: array<AbsDom>, ?cl: bool, ?p: int, ?v: AbsDom, ?cdata:
      CallData]
 MState{!id, !pc}(?size, ?sa , ?mem, ?stor, ?cl, ?cdata), ?size > 0,
 !a != ~1, // in case that the position could be pre-computed, use it for accessing the position more precisely
 ?v = accessWordCalldata{!a}(?cdata) // accesses word at the corresponding position of the call data
 => MState{!id, !pc +1}(?size, store ?sa (?size -1) (?v), ?mem, ?stor, ?cl, ?cdata),
clause [?x: AbsDom, ?size: int, ?sa: array<AbsDom>, ?mem: array<AbsDom>, ?stor: array<AbsDom>, ?cl: bool, ?cdata: CallData, ?p: int, ?v:
       AbsDom]
 MState{!id, !pc}(?size, ?sa , ?mem, ?stor, ?cl, ?cdata), ?size > 0,
 !a = ~1, // if the argument could not be preecomputed, extract the argument from stack
 ?x = select ?sa (?size - 1),
 ?v = (isConcrete(?x)) ? (accessWordCalldataEven(extractConcrete(?x), ?cdata)) : (@T) // if the offset is concrete, try to access the
       word at the given position. This will only result in a concrete result if the value is a word position
 => MState{!id, !pc +1}(?size, store ?sa (?size -1) (?v), ?mem, ?stor, ?cl, ?cdata);
```

Figure 15: Rule for CALLDATALOAD in the enhanced abstract semantics.

```
rule opHaltOnReturn :=
    for (!id: int) in ids(), (!pc: int) in pcsForIdAndOpcode(!id, RETURN)
    let
      macro #StackSizeCheck := MState{!id,!pc}(?size, ?sa, ?mem, ?stor, ?cl, ?cdata), ?size > 1
    in
    clause [?sa: array<AbsDom>, ?mem: array<AbsDom>, ?stor: array<AbsDom>, ?size:int, ?cl: bool, ?cdata: CallData, ?length: AbsDom]
        #StackSizeCheck, ?length = select ?sa (?size-2)
        => Halt{!id}(?stor, ?length, ?cl, ?cdata),
    clause [?sa: array<AbsDom>, ?mem: array<AbsDom>, ?stor: array<AbsDom>, ?size:int, ?cl: bool, ?offset: AbsDom, ?length: AbsDom, ?o:
          int, ?l:int, ?p:int, ?v: AbsDom, ?cdata: CallData]
        #StackSizeCheck, ?offset = select ?sa (?size-1), ?length = select ?sa (?size-2), // select top values on the stack
        isConcrete(?offset), isConcrete(?length),
        ?o = extractConcrete(?offset), ?l = extractConcrete(?length),
        ?p >= 0, (?p * 32) < ?l, // write all words that are still within the length
        ?v = accessWordMemoryEven(?o + ?p, ?mem)
        => ReturnData{!id}(?p, ?v, ?cl, ?cdata), // careful: the Return data predicate is also inhabited in words!
    clause [?sa: array<AbsDom>, ?mem: array<AbsDom>, ?stor: array<AbsDom>, ?size:int, ?cl: bool, ?offset: AbsDom, ?length: AbsDom, ?o:
          int, ?l:int, ?p:int, ?v: AbsDom, ?cdata: CallData]
        #StackSizeCheck, ?offset = select ?sa (?size-1), ?length = select ?sa (?size-2), // select top values on the stack
        ~isConcrete(?offset), isConcrete(?length), // knowing only the length, we write top at the places in the specified range
        ?l = extractConcrete(?length), ?p >= 0, ?p * 32 < ?l
        => ReturnData{!id}(?p, @T, ?cl, ?cdata),
    clause [?sa: array<AbsDom>, ?mem: array<AbsDom>, ?stor: array<AbsDom>, ?size:int, ?cl: bool, ?offset: AbsDom, ?length: AbsDom, ?o:
          int, ?l:int, ?p:int, ?v: AbsDom, ?cdata: CallData]
        #StackSizeCheck, ?length = select ?sa (?size-2), ~isConcrete(?length), ?p >= 0
        => ReturnData{!id}(?p, @T, ?cl, ?cdata);
```

Figure 16: Rule for RETURN in the enhanced abstract semantics.

RETURN semantics is closely reflected in the abstract RETURN rule: the first clause of the rule inhabits the `Halt` predicate, reading the size of the return data from the stack. The next three clauses inhabit the `ReturnData` predicate, differentiating depending on how much information on the return data (size and memory offset) are known: If both memory offset and length of the data are known, for each word position `?p` the corresponding memory word is read from the memory array `?mem` (using the function `accessWordMemoryEven`) and written into the `ReturnData` predicate. The next clause describes the case where the memory offset is unknown, but the size of the return data is known. In this case we cannot know which (concrete values) form the return data, but can only approximate all possible return data words (as determined by the size of the array) with `@T`. The last clause covers the case where the length of the return data is unknown. Since it is unclear in this case whether data should be returned in the first place (since the length could be 0), all potential positions of the return data array are over-approximated by `@T`.

Finally, the functional correctness queries for the addition function of the SafeMath library are depicted in Figure 17. We first specify the call data for a call to the add function of the SafeMath library as an operation `callAdd` returning an `CallData` element when being provided with the arguments to the call. Since the add function expects 2 integer arguments the `callAdd` function returns call data of size 68 (4+2∗32) bytes where the `@T` array is initialized with the hash of the corresponding function signature as first element (which represents the first 4 bytes of the call data) and the arguments `x` and `y` as following two elements. Note that the hash of the function signature is provided by Solidity compilers via the so called Ethereum Contract ABI (Contract Application Binary Interface). We plan to automatically generate an infrastructure for functional correctness queries on Solidity contracts from the contract ABI. The first functional correctness test `addOverflowNoHalt` requires that it is impossible to reach a `Halt` state (indicating regular halting) from a call to the add function in case that the addition of `?x` and `?y` overflows.

The second functional correctness test (`addNoOverflowCorrect`) checks whether it is possible (in case that no overflow occurs) to compute the expected result (or an over-approximation thereof) in the first place. Here `abseq` is the function implementing an equality test on the abstract domain, hence considering every concrete element to be potentially equal to `@T`. By the soundness of the analysis, if this query would turn out to be unsatisfiable, it would be impossible

```
op callAdd(x: int, y: int): CallData :=
 @D(68, store (store (store [@T] 0 @V(1997931255)) 1 (@V(x))) 2
        (@V(y)));

test addOverflowNoHalt expect UNSAT
  for (!id: int) in ids()
  [?x:int, ?y: int, ?z:int, ?p:int, ?stor: array<AbsDom>, ?
        rdsize:AbsDom]
   ?x >= 0, ?y >= 0, ?x < MAX, ?y < MAX, ?x + ?y >= MAX,
   Halt{!id}(?stor, ?rdsize, false, callAdd(?x, ?y));

test addNoOverflowCorrect expect SAT
  for (!id: int) in ids()
  [?res: AbsDom, ?x:int, ?y: int, ?z:int, ?rdsize:AbsDom, ?stor
        : array<AbsDom>]
   ?x >= 0, ?y >= 0, ?x + ?y < MAX,
   ReturnData{!id}(0, ?res, false, callAdd(?x, ?y)),
   Halt{!id}(?stor, ?rdsize, false, callAdd(?x, ?y)),
   abseq(?rdsize, @V(32)), abseq(?res, @V(?x + ?y));

test addNoOverflowHalt expect UNSAT
  for (!id: int) in ids()
  [?res: AbsDom, ?x:int, ?y: int, ?z:int, ?rdsize: AbsDom, ?
        stor: array<AbsDom>]
  ?x >= 0, ?y >= 0, ?x + ?y < MAX,
  Halt{!id}(?stor, ?rdsize, false, callAdd(?x, ?y)),
  ?rdsize != @V(32);

test addNoOverflowUnique expect UNSAT
  for (!id: int) in ids()
  [?res: AbsDom, ?x:int, ?y: int, ?z:int, ?rdsize: AbsDom, ?
        stor: array<AbsDom>]
    ?x >= 0, ?y >= 0, ?x + ?y < MAX,
    ReturnData{!id}(0, ?res, false, callAdd(?x, ?y)),
    ?res != @V(?x + ?y);
```

Figure 17: Correctness queries for SafeMath's add function

```
sel preStorageForId: int -> [int*int];
sel postStorageForId: int -> [int*int];
sel emptyListIfNoPostConditionForId: int -> [bool];
sel dummyListIfNoPostConditionForId: int -> [bool];

rule initOp :=
  for (!id:int) in ids()
  clause
    true
   => MState{!id, 0}(0, [@V(0)], [@V(0)],
      for (!offset: int, !value:int) in preStorageForId (!id):
       x: array<AbsDom> -> store x !offset @V(!value), [@V(0)],
      false);

test correctValues expect SAT
  for (!id: int) in ids(),
      (!b: bool) in emptyListIfNoPostConditionForId(!id)
  [?stor: array<AbsDom>, ?i: int]
   for (!offset: int, !value:int) in postStorageForId(!id):
    && abseq(select ?stor !offset,@V(!value)),
   (for (!offset: int, !value:int) in postStorageForId(!id):
    || ?i = !offset) ? (true) : (abseq(select ?stor ?i,@V(0))),
   Halt{!id}(?stor, false);

test uniqueValues expect UNSAT
  for (!id: int) in ids(),
      (!b: bool) in emptyListIfNoPostConditionForId(!id)
  [?stor: array<AbsDom>]
   for (!offset: int, !value:int) in postStorageForId(!id):
    || absneq(select ?stor !offset,@V(!value)),
   Halt{!id}(?stor, false);

test irregularHalt expect UNSAT
  for (!id: int) in ids(),
       (!b: bool) in dummyListIfNoPostConditionForId(!id)
  [?stor: array<AbsDom>]
   Halt{!id}(?stor, false);
```

Figure 18: Setup for automated testing.

for the function to produce the correct result under any circumstances. This query of course does not prove that the function will always provide a result: This indeed is and should not be provable, since any smart contract can always halt exceptionally when running out of gas. This test case serves as a sanity check that only becomes meaningful in conjunction with the following tests. The third and fourth functional correctness tests (`addNoOverflowHalt` and `addNoOverflowUnique`) prove that given non-overflowing arguments, if the function execution halts successfully, nothing but the correct result can be produced. In other words, it is impossible to halt successfully without producing the correct result. This property is composed of two queries since it needs to be shown that 1) It is impossible for the function to halt without returning a result of length 32 (corresponding to one word) as recorded in the `Halt` predicate and 2) It is impossible that the actual return value (as recorded in the `ReturnData` predicate) differs from the sum of the arguments.

The functional correctness tests for the other functions of the SafeMath library follow the same pattern.

C.3 Automated Testing in *HoRSt*

The setup for automated testing (see § 5.3) shown in Figure 18 presents a use case for the Hoare-Logic-style reasoning capabilities of *eThor*.

Test cases in the official test suite come in two flavors: the first group consists of 490 test cases specifying a storage configuration as postcondition, the second group, consisting of 108 test cases, lacks a postcondition (which we interpret as exceptional halting).

To account for this test structure we declare four additional selector functions: The selector functions `preStorageForId` and `postStorageForId` provide tuples of storage offsets and values which specify the storage contents before and after the execution of the contract. `emptyListIfNoPostConditionForId` and `dummyListIfNoPostConditionForId` generate an empty list, respectively a list with one element, depending on if there is a postcondition specified or not. Since rules are generated for the cross product of their selector functions return values, we can use these functions to generate different rules for different test cases while still using the same *HoRSt* inputs.

The rule for initialization, `initOp`, slightly differs from the definition used in the other experiments. The analysis is initialized to start in a storage as specified by `preStorageForId`.

To check for the reachability of a certain storage configuration, we generate the two queries `correctValues` and `uniqueValues`. `correctValues` is successful, if a `Halt` predicate is reachable whose storage contains 1) values abstractly equal to the values returned by `postStorageForId` at the offsets returned by `postStorage` and 2) a value abstractly equal to 0 for all offsets not returned by `postStorageForId`. `uniqueValues` is successful, if no `Halt` predicate is reachable whose storage contains any value abstractly unequal to the values returned by `postStorageForId`. This is only the case, if every value in the memory is concrete. In sum, such a test case is considered to be solved *correctly* if `correctValues` is successful and considered to be solved *precisely* if `correctValues` and `uniqueValues` are successful. To check for exceptional halting, we just query for the unreachability of a regular `Halt` predicate (see `irregularHalt`). Such a query is considered solved precisely on success and imprecisely on failure, since reaching additional program states (`Halt` in this instance), which are not reachable in the concrete execution, indicates over-approximation.