EasyChair Preprint

№ 10335

HoRStify: Sound Security Analysis of Smart Contracts

Sebastian Holler, Sebastian Biewer and Clara Schneidewind

June 5, 2023

HoRStify: Sound Security Analysis of Smart Contracts *

Sebastian Holler[1], Sebastian Biewer[2], and Clara Schneidewind[1]

[1] Max-Planck-Institute for Security & Privacy, Universitätsstraße, Bochum, Germany
{holler, schneidewind}@mpi-sp.org
[2] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
biewer@depend.cs.uni-saarland.de

Abstract

The cryptocurrency Ethereum is the most widely used execution platform for smart contracts. Smart contracts are distributed applications, which govern financial assets and, hence, can implement advanced financial instruments, such as decentralized exchanges or autonomous organizations (DAOs). Their financial nature makes smart contracts an attractive attack target, as demonstrated by numerous exploits on popular contracts resulting in financial damage of millions of dollars. This omnipresent attack hazard motivates the need for sound static analysis tools, which assist smart contract developers in eliminating contract vulnerabilities a priori to deployment.

Vulnerability assessment that is sound and insightful for EVM contracts is a formidable challenge because contracts execute low-level bytecode in a largely unknown and potentially hostile execution environment. So far, there exists no provably sound automated analyzer that allows for the verification of security properties based on program dependencies, even though prevalent attack classes fall into this category. In this work, we present HoRStify, the first automated analyzer for dependency properties of Ethereum smart contracts based on sound static analysis. HoRStify grounds its soundness proof on a formal proof framework for static program slicing that we instantiate to the semantics of EVM bytecode. We demonstrate that HoRStify is flexible enough to soundly verify the absence of famous attack classes such as timestamp dependency and, at the same time, performant enough to analyze real-world smart contracts.

1 Introduction

Modern cryptocurrencies enable mutually mistrusting users to conduct financial operations without relying on a central trusted authority. Foremost, the cryptocurrency Ethereum supports the trustless execution of arbitrary quasi Turing-complete programs, so-called smart contracts [28], which manage money in the virtual currency Ether.

The expressiveness of smart contracts gives rise to a whole distributed financial ecosystem known as Decentralized Finance (DeFi), which encompasses a multitude of (financial) applications such as brokerages [19,29], decentralized exchanges [2,15,30] or decentralized autonomous organizations [12,25]. However, smart contracts have shown to be particularly prone to programming errors that lead to devastating financial losses [4]. These severe incidents can be attributed to different factors. First, smart contracts are agents that interact with a widely

*Presented at 24th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-24) [13]. This work has been supported by the Heinz Nixdorf Foundation through a Heinz Nixdorf Research Group (HN-RG) and funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy—EXC 2092 CASA—390781972, and through grant 389792660 as part of TRR 248—CPEC, see https://perspicuous-computing.science.

unpredictable and potentially hostile environment. Accounting for all possible environment behaviors adds a layer of complexity to smart contract development. Second, smart contracts manage real money. This financial nature makes them an extraordinarily lucrative attack target. Third, transactions in blockchain-based cryptocurrencies, like Ethereum, are inherently immutable. As a consequence, not only the effects of exploits are persistent, but also vulnerable smart contracts cannot be patched. Given this state of affairs, it is of utmost importance to preempt contract vulnerabilities a priori to contract deployment.

Sound static analysis tools allow for reasoning about all possible runtime behaviors without deploying a contract on the blockchain. In this way, smart contract developers and users can reliably identify and eliminate harmful behavior before publishing or interacting with Ethereum smart contracts. However, as shown in recent works [23,24], most automatic static analyzers for Ethereum smart contracts that promise soundness guarantees cannot live up to their soundness claims.

To the best of our knowledge, the only tools targeting sound and automated static analyses of smart contract security properties are Securify [26], ZEUS [17], EtherTrust [10], NeuCheck [20], and eThor [23]. The soundness claims of ZEUS, Securify, EtherTrust, and NeuCheck are systematically confuted in [24] and [23].

The analysis tool eThor [23] comes with a rigorous soundness proof but only supports the verification of reachability properties. While this is sufficient to characterize the absence of interesting attack classes, many other smart contract security properties do not fall within this property fragment. Grishenko et al. [11] give a semantic characterization of security properties that characterize the absence of prominent classes of smart contract bugs. Most of these properties fall into the class of non-interference-style two-safety properties that we will refer to as *dependency properties* and fall out of the scope of eThor's analysis. The only tool that, up to now, targeted the (sound) verification of dependency properties was the tool Securify [26]—which was empirically shown unsound in [23].

Our Contributions In this work, we revisit Securify's approach. In this course, we analyze the peculiar challenges in designing a sound static dependency analysis tool for Ethereum smart contracts. We show how to overcome these obstacles with a principled approach based on rigorous formal foundations. Leveraging a formal proof framework for static program slicing [27], we design a provably sound dependency analysis for Ethereum smart contracts on the level of Ethereum Virtual Machine (EVM) bytecode. Finally, we give an implementation of the analyzer HoRStify that performs the static dependency analysis via a logical encoding, which can be automatically solved by Datalog solvers. We demonstrate how to use HoRStify to automatically verify dependency properties on smart contracts, such as the ones defined in [11]. Concretely, we make the following contributions: (i) We devise a new dependency analysis for EVM bytecode based on program slicing following the static program framework presented in [27]. (ii) We prove this dependency analysis to be sound with respect to a formal semantics of EVM bytecode. (iii) We show how to approach relevant smart contract security properties presented in [11] with the dependency analysis. (iv) We present HoRStify, an automated prototype static analysis tool that implements the dependency analysis. (v) We demonstrate that HoRStify overcomes the soundness issues of Securify while showing comparable performance and small precision loss on real-world smart contracts.

The remainder of the paper is organized as follows: Section 2 overviews our approach and discusses the challenges in designing sound static analysis tools for smart contract dependency analysis; Section 3 introduces the necessary background on Ethereum smart contract execution and the current state of smart contract dependency analysis; Section 4 introduces the

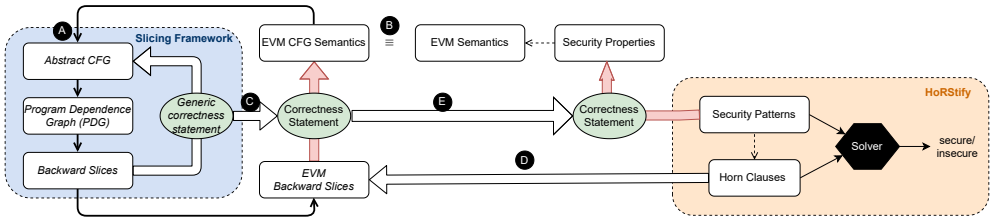

Figure 1: Overview on the formal guarantees of HoRStify

slicing proof framework from [27] that our analysis builds on; Section 5 presents our static analysis based on program slicing and its soundness proof; Section 6 reports on our prototype implementation HoRStify and its practical evaluation; and Section 8 concludes the paper.

2 Overview

In this paper, we develop a dependency analysis tool for EVM bytecode that is designed in accordance with formal correctness statements providing overall soundness guarantees. The correctness proof is modularized as depicted in Figure 1. The core module is a generic proof framework [27] for backward slicing using abstract control flow graphs (CFGs). In these CFGs, each node is annotated with all variables it reads and all variables it writes. The backward slice of a node is a set containing all nodes that possibly influence the variables written in the respective node. The framework extends the abstract CFG to a program dependence graph (PDG) by explicitly defining the data and control dependencies between the nodes. For this PDG, the framework establishes a generic correctness statement for slicing: whenever a node influences another, the influencing node appears in the backward slice of the influenced node. To obtain the correctness result for a concrete programming language the abstract CFG representation is instantiated for a concrete program semantics.

We instantiate the framework for EVM bytecode by devising a new EVM CFG semantics. We show (A) that the EVM CFG semantics satisfies all requirements for instantiation and (B) that it is equivalent to a formalisation of the EVM bytecode semantics. From this, we obtain backward slicing for EVM contracts with a corresponding correctness statement (C).

For the actual analysis, we express dependencies in EVM contracts by means of dependency predicates which we characterize by (fixpoints over) a set of logical rules, given in the form of *Constrained Horn Clauses (CHC)*. Most importantly, we show that if the backward slice of some program point contains some other program point, then the (potential) dependency between these two program points is also captured by the predicate encoding (D).

From the EVM bytecode analyzer Securify [26] we adopt the idea of defining so-called security patterns to soundly approximate the satisfaction (or violation) of a security property. A *security pattern* is a set of facts over dependency predicates, which characterize the form of dependencies that are ruled out by the pattern. In contrast to Securify, our formal characterization of dependency predicates enables a correctness statement for the approximating behavior of the security patterns w.r.t. their corresponding property (E).

Finally, we present the prototype tool HoRStify that implements our dependency analysis and uses the Datalog engine Soufflé to perform the fixpoint computation and to check whether a security pattern is matched. A pattern match guarantees (in)security w.r.t. the respective security property.

Challenges The main challenge of designing a practical and sound dependency analysis for EVM bytecode is finding precise and performant abstractions that tame the complexity of EVM bytecode while maintaining soundness guarantees. As we have elaborated in [14], EVM bytecode's language design makes this task particularly hard: Non-standard language features introduce corner cases that are easily overlooked or make it necessary to enhance the analysis with custom optimizations that can lead to unsoundness when done in an ad-hoc manner. As a consequence, it is of paramount importance to construct a sound analysis tool with formal foundations that are flexible enough to cover those subtleties.

The slicing framework [27] enables a modular soundness proof that separates the standard argument for the correctness of slicing from the characterization of program dependencies. However, even though this reduces the proof effort, a naive instantiation of the framework would introduce a multitude of superfluous dependencies and hence lead to a highly imprecise analysis. For this reason, the key challenge lies in the design of the EVM CFG semantics. We will show how to approach these challenges with a solid theoretical foundation and by circumventing the bothersome technical hurdles without compromising the soundness of the analysis. In this paper, we give a high-level overview of the relevant theorems and proofs and refer the interested reader to an extended version of this paper [14] for the technical details.

3 Background on Ethereum Smart Contracts Analysis

Ethereum smart contracts are distributed applications that are jointly executed by the users of the Ethereum blockchain. In the following, we shortly overview the workings of Ethereum, the resulting particularities of the Ethereum smart contract execution environment and introduce the smart contract dependency analysis tool Securify.

Ethereum The cryptocurrency Ethereum supports smart contracts via an account-based execution model. The global state of the system is given by accounts whose states are modified through the execution of transactions. All accounts have in common that they hold a balance in the currency Ether. An account can be either an *external account* that is owned by a user of the system and that solely supports user-authorized money transfers, or a *contract account* that manages its spending behavior autonomously by means of a program associated with the contract that may use its own persistent storage to provide advanced stateful functionalities.

Users interact with accounts via transactions. Transactions either call existing accounts or create new contract accounts. A call transaction transfers an amount of money (that could be 0) to the target account and triggers the execution of the account's code if the target is a contract account. A contract execution can modify the contract's persistent storage and potentially initiates further transactions. In this case, we speak of *internal* transactions, as opposed to *external* transactions, which are initiated by users on behalf of external accounts.

Smart Contract Languages Smart contracts are specified in *EVM bytecode* and executed by the *Ethereum Virtual Machine* (EVM). EVM bytecode is a stack-based low-level language that supports standard instructions for stack manipulation, arithmetics, jumps, and memory access. On top, EVM's instruction set includes blockchain-specific opcodes, for example, to access transaction information and to initiate internal transactions. While the EVM bytecode is technically Turing-complete, the execution of smart contracts is bounded by a transaction-specific resource limit. With each transaction, the originator sets this limit in the unit *gas* and pays for it upfront. During the execution, instructions consume gas. The execution halts with an exception if running out of gas and reverts all effects of the prior execution.

In practice, Ethereum smart contracts are written in high-level languages—foremost, Solidity [1]—and compiled to EVM bytecode. Solidity is an imperative language that mimics features of object-oriented languages like Java but supports additional primitives for accessing blockchain information and performing transactions. For better readability, we will give examples using the Solidity syntax even though our analysis operates on EVM bytecode.

Adversarial Execution Environment The blockchain environment poses novel challenges to the programmers of smart contracts. As opposed to programs that run locally, smart contracts are executed in an untrusted environment. This means, in particular, that certain system parameters cannot be fully trusted. A prominent example of this issue is Ethereum's block timestamp: In Ethereum's blockchain-based consensus mechanism, the system is advanced by appending a bulk of transactions grouped into a block to the blockchain, a distributed tamper-resistant data structure. These blocks are created by special system users, so-called miners. While all system users check that blocks only contain valid transactions, the correctness of a block's metadata cannot easily be verified. So is each block required to carry a timestamp, but due to the lack of synchronicity in the system, this timestamp can only be checked to lie within a plausible range. This enables a miner to choose the value of the block timestamp freely within this range. Therefore, the block timestamp should not be used as a source of randomness as illustrated in the following example: `function rouletteWheel() private (uint) { return timestamp % 37; }`

Securify The automated analyzer Securify is the only analysis tool up to now that aims at giving provable guarantees for dependency analyses of EVM bytecode contracts. It decompiles the bytecode into a stackless *intermediate representation* (IR), where values are stored in variables in static single assignment (SSA) form rather than on a stack. Further, it determines the CFG of the contract and encodes the transitive control and data flow dependencies between variables and program locations as a set of *dependency predicates*. While it is not possible to specify arbitrary (security) properties in Securify, the tool allows for defining *compliance patterns* and *violation patterns* that serve as "approximations" for the satisfaction and, respectively, the violation of the property. These patterns are defined over the dependency predicates and can be checked automatically using the Datalog solver Soufflé [16]. A *compliance (violation) pattern is sound w.r.t. a property*, if satisfying the pattern implies satisfaction (violation) of the property. If neither of the patterns is satisfied, the satisfaction of the property is inconclusive.

An example of a security property is the *restricted write* (RW) property for storage locations. Intuitively, a contract satisfies RW, if for all storage locations, there is at least one caller address that cannot write to this location. We have have demonstrated in [14] how the lack of formal foundations affects the guarantees of the state-of-the-art analysis tool Securify [26].

4 Analysis Foundations

To design a sound static analysis for EVM bytecode based on program slicing, we instantiate the slicing proof framework from [27] with a formal bytecode semantics as defined in [11]. Before discussing the instantiation in Section 5, we shortly overview both frameworks.

4.1 EVM bytecode semantics

The EVM semantics was formally defined in [11] in form of a small-step semantics. We use a *linearized* representation of the semantics inspired by Securify, where the use of the stack is

replaced by the usage of local variables in SSA form. We will call these variables *stack variables* and, in the following, always refer to the linearized representation of the semantics.

Formally, the semantics of EVM bytecode is given by a small-step relation $\Gamma \vDash S \rightarrow S'$. The relation describes how a contract, whose execution state is given by a callstack S, can progress to callstack S' under a transaction environment Γ. The transaction environment Γ holds information about the external transaction that initiated execution. We let $\Gamma \vDash S \rightarrow^* S'$ denote the reflexive transitive closure of the small-step relation and call the pair (Γ, S) a *configuration*. The details of the components of the EVM configurations can be found in [11]. The overall state of an external transaction execution is captured by a callstack S. The elements of the callstack model the states of all (pending) internal transactions. Internal transactions can either be pending, as indicated by a regular execution state (μ, ι, σ), or terminated. The state of a pending transaction encompasses, the current global state σ, the execution environment ι and the machine state μ. The global state σ describes the state of all accounts of the system and is defined as a partial mapping between account addresses and account states. The execution environment ι, among others, contains the *code* of the currently executing contract. We model the code of a contract as a function C that maps program counters to tuples $(op(\vec{x}), pc_{next}, pre)$, where op denotes an opcode from the EVM instruction set, $\vec{x}$ is the vector of input and output (stack) variables to this opcode, and pc_{next} denotes the program counter for the next instruction. Further, we instrument each instruction with a list pre of precomputed values for the arguments $\vec{x}$. This instrumentation is only introduced for analysis purposes and does not affect the execution.

The machine state μ captures the state of the local machine and holds the amount of gas (g) available for execution, the program counter (pc), the local memory, and the state of the (linearized) stack variables (s).

Small-step Rules We illustrate the working of the EVM bytecode semantics using the example of the ADD instruction. This instruction takes two values as input and writes their sum back to its return variable.

$$\frac{\begin{array}{c}\iota.code\,[\mu.\mathsf{pc}] = (\mathsf{ADD}(r,a,b), pc_{next}, pre) \\ \mu.\mathsf{g} \geq 3 \qquad \mu' = \mu[\mathsf{s} \rightarrow \mu.\mathsf{s}[r \rightarrow \mu.\mathsf{s}(a) + \mu.\mathsf{s}(b)]][\mathsf{pc} \rightarrow pc_{next}][\mathsf{g} -= 3]\end{array}}{\Gamma \vDash (\mu, \iota, \sigma) :: S \xrightarrow{\mathsf{ADD}(a,b)} (\mu', \iota, \sigma) :: S}$$

Given a sufficient amount of gas (here 3 units), an ADD instruction with result (stack) variable r and operand (stack) variables a and b writes the sum of the values of a and b to r and advances the program counter to pc_{next}. These effects, as well as the subtraction of the gas cost, are reflected in the updated machine state μ'.

Security properties Previous work [11] has shown that there are several generic smart contract security properties, which are desirable irrespective of the individual contract logic. The properties formally defined in [11] are integrity properties that aim at ruling out the influence of attacker behavior on sensitive contract actions, in particular, the spending of money. These properties are e.g., the independence of a contract's spending behavior from miner-controlled parameters (as the block timestamp) or mutable contract state. Further, [11] introduces the notion of call integrity, which requires that the spending behavior of a contract is independent of the code of other smart contracts. Since call integrity is hard to verify in the presence of reentering exeutions, a proof strategy is devised that decomposes call integrity into one reachability property (single-entrancy) that restricts reentering executions and two local dependency properties. These local dependency properties ensure that the spending behavior of the contract

does not depend on the return effects of calls to other (unknown) contracts (effect independence) or immediately on the code of such contracts (code independence).

Focussing on integrity, the security properties from [11] are given as non-interference-style notions. We illustrate this with the example of timestamp independence, a property that requires that the block timestamp cannot influence a contract's spending behavior and hence would rule out vulnerabilities as those in the roulette example:

Definition 1 (Independence of the block timestamp)**.** *A contract C is independent of the block timestamp if for all reachable configurations $(\Gamma, s_C :: S)$ it holds for all Γ' that*

$$\Gamma =_{/timestamp} \Gamma' \ \wedge \Gamma \vDash s_C :: S \xrightarrow{\pi}{}^{*} s'_C :: S \ \wedge \ \mathit{final}(s') \\ \wedge \ \Gamma' \vDash s_c :: S \xrightarrow{\pi'}{}^{*} s''_C :: S \ \wedge \ \mathit{final}(s'') \implies \pi \downarrow_{\mathsf{calls}_C} = \pi' \downarrow_{\mathsf{calls}_C}$$

This definition requires that two executions of the contract C starting in the same execution state s_C and in transaction environments Γ and Γ' that are equal up to the block timestamp (denoted by $\Gamma =_{/timestamp} \Gamma'$) exhibit the same calling behavior (captured by the call traces $\pi \downarrow_{\mathsf{calls}_C}$). Intuitively, this ensures that the contract C may not perform different money transfers based on the block timestamp. The roulette example trivially violates this property since, based on the block timestamp, the prize will be paid out to a different user.

4.2 Program Slicing

Static program slicing is a method for capturing the dependencies between different program points (nodes) and variables in a program. Intuitively, the program slice of some program node n in a program P consists of all those nodes n' in P that may affect the values of variables written in n. Program slices are constructed based on the program dependence graph (PDG) that models the control and data dependencies between the nodes of a program. In the following, we will review the static slicing framework by Wasserraab et al. [27], which establishes a language-independent correctness result for slicing based on abstract control flow graphs (CFGs).

Abstract control flow graph An abstract CFG is a language-agnostic representation of program semantics. Technically, an abstract CFG is parametrized by a set of program states Θ and defined by a set of nodes (representing program points) and a set of directed edges between nodes. Edges may be of two different types: State-changing edges $n - \Uparrow f \rightarrow n'$ alter the program state $\theta \in \Theta$ by applying the function f to θ and predicate edges $n - (Q)_{\surd} \rightarrow n'$ guard the transition between n and n' with the predicate Q on the program state θ. We write $n \xrightarrow{as}{}^{*} n'$ to denote that node n can be reached n' using the edges in the list as. Abstract CFG edges can be related to actual runs of the program by lifting them to a small-step relation of the form $\langle n,\ \theta \rangle \ -a\rightarrow \langle n',\ \theta' \rangle$.

PDG and backward slices The PDG for a program consists of the same nodes as the CFG for this program and has edges that indicate data and control dependencies. To make data dependencies inferable, each node n is annotated with a set of variables that are written (short *Def set*, written $\mathit{Def}(n)$) and a set of variables that are read by the outgoing edges of the node (short *Use set*, written $\mathit{Use}(n)$). A node n' is data dependent on node n (written $n \rightarrow_{dd} n'$) if n defines a variable Y ($Y \in \mathit{Def}(n)$), which is used by n' ($Y \in \mathit{Use}(n')$) and n' is reachable from n in the CFG without passing another node that defines Y. A node n' is (standard) control dependent on node n (written $n \rightarrow_{cd} n'$) if n' is reachable from n in the CFG, but n can as well

reach the program's exit node without passing through n' and all other nodes on the path from n to n' cannot reach the exit node without passing through n'. So intuitively, n is the node at which the decision is made whether n' will be executed or not. Based on the data and control flow edges of the PDG, the backward slice of a node n (written $BS(n)$) is defined as the set of all nodes n' that can reach n within the PDG.

Correctness statement The generic correctness statement for slicing proven in [27] is stated as follows:

Theorem 1. *Correctness of Slicing Based on Paths [27]*

$$\frac{\langle n,\ \theta\rangle \xrightarrow{as}{}^{*} \langle n',\ \theta'\rangle}{\exists\ as'.\ \langle n,\ \theta'\rangle \xrightarrow{as'}{}^{*}_{BS(n')} \langle n',\ \theta''\rangle\ \wedge\ as\downarrow_{BS(n')} = as'\ \wedge\ (\forall\ V \in Use(n').\theta'(V) = \theta''(V))}$$

Intuitively, the theorem states that whenever a node n can reach some node n' in the PDG ($\langle n,\ \theta\rangle \xrightarrow{as}{}^{*} \langle n',\ \theta'\rangle$), then removing all outgoing edges from nodes not in the backward slice of n' ($\langle n,\ \theta\rangle \xrightarrow{as'}{}^{*}_{BS(n')} \langle n',\ \theta''\rangle$) without altering the path through the PDG in any other way ($as\downarrow_{BS(n')} = as'$) has no impact on n'. Having no impact on n' means that variables used in n' are assigned to the same values regardless of whether the edges have been removed or not ($\forall V \in Use(n').\ \theta'(V) = \theta''(V)$). We call the PDG without the above-mentioned edges also *sliced PDG* or *sliced graph*.

5 Sound EVM Dependency Analysis

Next, we instantiate the slicing proof framework [27] to accurately capture program dependencies of EVM smart contracts in terms of program slices. We then give a logical characterization of such program slices, which allows for the automatic computation of dependencies between different program points and variables with the help of a Datalog solver. The generic correctness statement of the slicing proof framework guarantees that the slicing-based dependencies soundly over-approximate all real program dependencies. We show how to use this result to automatically verify relevant smart contract security properties such as the independence of the transaction environment and the independence of mutable account state as defined in [11].

5.1 Instantiation of Slicing Proof Framework

We instantiate the abstract CFG from the slicing framework with the linearized EVM semantics.

The concrete layout of the instantiation heavily influences the resulting backward slices and the precision of the analysis. In the following, we sketch the most interesting aspects of our instantiation of the CFG components and how they contribute to the design of a precise dependency analysis.

Preprocessing Information For a precise analysis, it is indispensable to preprocess contracts to aggregate as much statically obtainable information as possible—without compromising the soundness of the overall analysis. For example, knowing the precise destination of jump instructions is crucial to reconstruct control flow precisely, and, moreover, this information usually can be easily reconstructed, especially, when contracts were compiled from a high-level language with structured control flow.

In the remainder, we assume that all existing preprocessing information is correct and sufficient to reconstruct the contract's CFG. Recall that, formally, we consider a contract a function, such that for a program counter pc, $C(pc) = (op(\vec{x}), pc_{\text{next}}, pre)$ where pre contains the preprocessing information for the instruction $op(\vec{x})$: for every $\vec{x}[i]$, $pre[i]$ either holds a precomputed static value, or $\perp$ to indicate that no static value could be inferred. Note that we restrict preprocessing to stack variables. For our analysis, we are only interested in precomputed values for memory and storage locations and jump destinations.

CFG States The edges of the CFG are labeled with state-changing functions or predicates on states. For EVM bytecode programs, the CFG state θ is partitioned into stack variables (denoted by x^{ls}), memory variables (x^m), storage variables (x^g) and local (x^{el}) and global (x^{eg}) environmental variables. Memory and storage variables represent cells in the local memory, respectively the global storage of the contract under analysis. Local environment variables contain the information of the execution environment that is specific to an internal transaction. Global environmental variables denote environmental information whose accessibility is not limited to a single internal transaction, like the state of other contracts and the block timestamp.

CFG Nodes, Edges & Def and Use Sets To transform an EVM bytecode program into a CFG, we map every program counter `pc` to one or more nodes (pc, i) in the CFG (where $i \in \mathbb{N}$ is used to distinguish between multiple nodes for pc). We call a node $(pc, 0)$ *initial node (for pc)* and nodes (pc, i) with $i > 0$ *intermediate nodes (for pc)*. Since the size of the callstack below the translated callstack element may influence the contract execution, the rule set defining the CFG transformation constructs a relation of the form $C, cd \vDash (pc, i) \; -a \rightarrow (pc', i')$, where C is the contract for which the CFG is constructed, cd is the size of the callstack, and a stands for either a $(Q)_{\surd}$ action (for a predicate edge) or $\Uparrow f$ action (for a state-changing edge). With every rule, we also provide Def and Use sets. The Use sets contain all variables whose values are retrieved from the state θ in the definition of the Q predicate or f function. Similarly, the definition set contains all variables that are overwritten by the function f (and is always empty for predicate edges).

It can be shown that the CFG semantics and EVM semantics coincide via two simulation relations where every (multi-)step in the CFG semantics between initial nodes is simulated by a step of the bytecode semantics and vice versa.

5.2 Core abstractions

Memory Abstraction To precisely model memory and storage accesses in a CFG, it is important to know statically as many memory and storage locations as possible. Assume that such statical information is not available: then memory (or storage) cannot be separated into regions and all read and write operations introduce dependencies with the whole memory (or storage). This would introduce many false dependencies. During a preprocessing step, such static information can be inferred. But, as illustrated by Securify, using preprocessed data may introduce unsoundness [14]. This requires careful integration of preprocessing information into the CFG defining rules. In the following we consider only memory variables; all ideas equally apply to storage variables.

We propose a, to the best of our knowledge, novel memory abstraction that is sound and provides high precision. To position our approach between unsound and imprecise memory abstractions, we investigate a contract satisfying the RW property depicted as a CFG in Figure 2. The black and solid line parts of the left CFG visualize how Securify misses the dependency

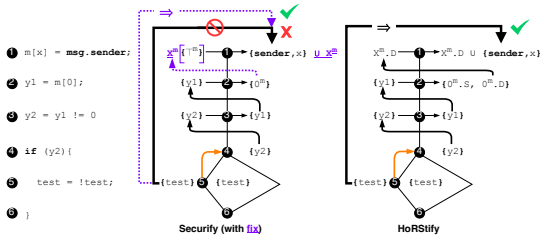

Figure 2: Contract satisfying the RW property with PDGs depicting the dependencies modeled by Securify and HoRStify.

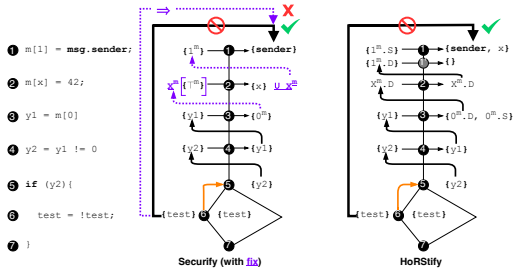

Figure 3: Contract violating the RW property with PDGs depicting the dependencies as modeled by Securify and HoRStify.

between `msg.sender` (❶) and writing to `test` (❺). In Securify, write accesses to unknown memory locations are assumed to write a special memory variable $\top^m$. However, when reading from a statically known memory location (as done in ❷), Securify does not consider that a value could have been written to this location when the location was not statically known, i.e., that the value could have been stored in $\top^m$: the Use set of ❷ contains only 0^m, but not $\top^m$. A hypothetical fix for this unsoundness is to replace the variable $\top^m$ by the whole set X^m of all memory variables. This fix is depicted in violet in Figure 2. Now, the dependency of the read access in ❷ to the write operation in ❶ is naturally established. One should notice, however, that this interpretation implies that the Use set of node ❶ needs to contain all variables in X^m as well: a new value is written to one unknown location, but for all other locations the value is "copied" from the existing memory cells, and hence, all these cells need to be included in the Use set. Even though fixing the soundness issue, this modeling would lead to an imprecise analysis as depicted in Figure 3. This variant of Figure 2 first writes `msg.sender` to the known memory location `1` in node ❶ and then writes a value to an unknown memory location in node ❷. Since the condition `y2` only depends on the value in memory location `0` while `msg.sender` was written to location `1`, the final write to the `test` variable in ❻ does **not** depend on `msg.sender`. However, the hypothetical fix of Securify infers a possible dependency between ❻ and `msg.sender` (shown in violet in the left CFG in Figure 3). This imprecision is caused by interpreting a write to an unknown memory location as a write to possibly all memory locations as this requires the Use set in ❷ to contain X^m. This creates a dependency between the assignment of location `1` to `msg.sender` in ❶ and the memory access in ❷.

Our memory abstraction is sound but more precise than the hypothetical fix above. For every memory variable x we use two sub-variables instead: *S-variable* $x^m.\mathsf{S}$ stores values that are assigned to x when the memory location for x is statically known, and *D-variable* $x^m.\mathsf{D}$ stores values assigned to x when x's location is not statically known. During the execution, every write access to a memory variable x stores the assigned value in $x^m.\mathsf{D}$, unless the memory location for x is statically known, in which case $x^m.\mathsf{S}$ stores the value and $x^m.\mathsf{D}$ is set to $\bot$. Correspondingly, when reading from a variable (regardless of the memory location being statically known or not), first, the value of the D-variable is read, and only if it is $\bot$, the value of the S-variable is taken. We model this read access with the function

$$load\ \theta\ x = \begin{cases} \theta[x^m.\mathsf{S}] & \text{if } \theta[x^m.\mathsf{D}] = \bot \\ \theta[x^m.\mathsf{D}] & \text{otherwise.} \end{cases}$$

This two-layered memory abstraction ensures that the execution is deterministic and that

the read values coincide with those obtained during an execution without prior preprocessing. The *load* function is used in the corresponding inference rule for memory operations that can be found in [14]. Figures 2 and 3 (right sides) showcases how this leads to a sound and precise dependency propagation.

Gas and Call Abstraction To incorporate a sound and precise modeling, we assume that a contract does not run out-of-gas and we are targeting only smart contracts that cannot write storage in reentering executions. These assumption can be checked statically [3, 23].

Assumption 1 (Absence of local out-of-gas exceptions (informal))**.** *A contract execution* does not exhibit local-out-of-gas exceptions *if each local exception can be attributed to the execution of an* INVALID *opcode.*

Assumption 2 (Store unreachability (informal))**.** *A contract C is store unreachable if all its reentering executions cannot reach an* SSTORE *instruction.*

Details about how the challenges of soundly and precisely modeling gas and calls in the CFG semantics are solved and why these assumption where made can be found in [14].

5.3 Soundness Reasoning via Dependency Predicates

Inspired by Securify, we define dependency predicates that can capture the data and control flow dependencies induced by the PDG (as given through the CFG semantics). They are inhabited via a set of logical rules (CHCs) $\mathcal{R}(C)$ that describe the data and control flow propagation through the PDG of a contract C. More formally, the transitive closure of the C's PDG is computed as the least fixed point over $\mathcal{R}(C)$ (denoted by $\mathit{lfp}(\mathcal{R}(C))$). Most prominently, $\mathit{lfp}(\mathcal{R}(C))$ includes the predicates $\mathit{VarMayDepOn}$ and $\mathit{InstMayDepOn}$. Intuitively, $\mathit{VarMayDepOn}(y, x)$ states that the value of variable y may depend on the value of variable x and $\mathit{InstMayDepOn}(n, x)$ says that the reachability of node n may depend on the value of variable x. In the following, let n_x and n_y denote nodes that define variables x and y, respectively. The formal relation between dependency predicates and backward slices is captured by the following lemma:

Lemma 1 (Fixpoint Characterization of Backward Slices)**.** *Let x and y be variables and C be a contract. The following holds:*

1. $(\exists n_x\ n_y.\ n_x \in \mathit{BS}(n_y)) \Rightarrow \mathit{VarMayDepOn}(y, x) \in \mathit{lfp}(\mathcal{R}(C))$
2. $(\exists\ n\ n_{if}\ n_x.\ n_{if} \rightarrow_{cd} n\ \wedge\ n_x \in \mathit{BS}(n_{if})) \Rightarrow \mathit{InstMayDepOn}(n, x) \in \mathit{lfp}(\mathcal{R}(C))$

Lemma 1 states 1) that whenever there is a node n_x defining x in the backward slice of a node n_y defining y, then $\mathit{VarMayDepOn}(y, x)$ is derivable from the CHCs in $\mathcal{R}(C)$ and 2) that whenever there is a node n_x defining x in the backward slice of a node n_{if} on which node n is control dependent then $\mathit{InstMayDepOn}(n, x)$ is derivable from $\mathcal{R}(C)$. The intuition behind statement 2) is that node n is controlled by n_{if} (by the definition of standard control dependence), which means that n_{if} is a branching node. $n_x \in \mathit{BS}(n_{if})$ indicates that the branching condition of n_{if} depends on variable x and, hence, so does the reachability of n.

Next, we give an explicit semantic characterization of the dependency predicates, which we prove sound using Theorem 1. This explicit characterization enables us to compose *security patterns* as a set of different facts over dependency predicates and to reason about them in a modular fashion. As a consequence, we can show in Section 5.4 that checking the inclusion of

security patterns in the least fixpoint of the rule set $\mathcal{R}(C)$ is sufficient to prove non-interference-style properties. Concretely, we can characterize facts from the *VarMayDepOn* predicate as follows:

Theorem 2 (Soundness of Dependency Predicates)**.**

$$\forall x\ y.\ \mathit{VarMayDepOn}(y,x) \notin \mathit{lfp}(\mathcal{R}(C)) \Rightarrow y \perp x \qquad \text{with } y \perp x \text{ given as:}$$

$$\forall n_x\ i\ \theta_1\ \theta_2\ \theta_1'.\ \theta_1 =_{/x} \theta_2\ \wedge\ \langle {n_x}^+, \theta_1\rangle \xrightarrow{N_y}{}^{i} \langle n, \theta_1'\rangle \Rightarrow \exists \theta_2'.\ \langle {n_x}^+, \theta_2\rangle \xrightarrow{N_y}{}^{i} \langle n, \theta_2'\rangle\ \wedge\ \theta_1'(y) = \theta_2'(y)$$

where ${n_x}^+$ *denotes the unique successor node of* n_x*, and* N_y *the set of all nodes defining* y*.* $\langle {n_x}^+, \theta_1\rangle \xrightarrow{N_y}{}^{i} \langle n, \theta_1'\rangle$ *describes an execution from* n_x *to* n *that passes exactly* i *nodes defining* y*.*

The theorem states that if $\mathit{VarMayDepOn}(y,x)$ is not included in $\mathit{lfp}(\mathcal{R}(C))$ then y is independent of x ($y \perp x$). A variable y is considered independent of x if for any two configurations θ_1 and θ_2 that are equal up to x, and any execution starting at node ${n_x}^+$, the first node after x is defined, passing i nodes that define y, and ending in a node n at state θ_1', one can find a matching execution from θ_2 that passes the same number of nodes defining y and ends at node n in a state θ_2' such that θ_2' and θ_1' agree on y. This definition ensures loop sensitivity: it captures that during a looping execution, every individual occurrence of a node defining y can be matched by the other execution—so that the values of y agree whenever y gets reassigned. The proof of Theorem 2 uses Lemma 1 and Theorem 1. For the full proof and a similar characterization of $\mathit{InstMayDepOn}(i,x)$, we refer to [14].

5.4 Sound Approximation of Security Properties

With Theorem 2 we are able to formally connect dependency predicates and (independence-based) security properties. We take *trace noninterference* as a concrete example, which comprises a whole class of non-interference-style security properties. Concretely, we consider trace noninterference w.r.t. a set of EVM configuration components Z, which includes, for example, the block timestamp. A predicate f defines *instructions of interest*. If two executions of a contract C start in configurations that differ only in the components in Z, then the instructions of interest must coincide in the two traces that result from these executions.

Definition 2 (Trace noninterference)**.** *Let* C *be an EVM contract,* Z *be a set of components of EVM configurations and* f *be a predicate on instructions. Then trace noninterference of contract* C *w.r.t.* Z *and* f *(written* $\mathit{TNI}(C,Z,f)$*) is defined as follows:*

$$\mathit{TNI}(C,Z,f) := \forall\ \Gamma\ \Gamma'\ s\ s'\ t\ t'\ \pi\ \pi'.\ ((\Gamma,s) =_{/Z} (\Gamma',s')) \quad \Rightarrow (\Gamma \vDash s_C :: S \xrightarrow{\pi}{}^{*} t_C :: S\ \wedge\ \mathit{final}(t))$$
$$\Rightarrow (\Gamma \vDash s_C' :: S \xrightarrow{\pi'}{}^{*} t_C' :: S\ \wedge\ \mathit{final}(t')) \qquad \Rightarrow \pi \downarrow_f = \pi' \downarrow_f$$

where $\pi \downarrow_f$ *denotes the trace filtered by* f*, so containing only the instructions satisfying* f*.*

The dependency properties defined in [11] can be expressed in terms of trace noninterference. E.g., the timestamp independence property in Definition 1 is captured as an instance of trace noninterference as follows:

$$\mathit{TNI}(C, \{\Gamma.\mathit{timestamp}\}, \lambda op.op = \mathsf{CALL})$$

We show that we can give a sufficient criterion for trace noninterference in terms of dependency predicates. More precisely, we give a set $\mathsf{P}^C_{\mathsf{Z},f}$ of facts, such that $\mathsf{P}^{\mathcal{R}(C)}_{\mathsf{Z},f} \cap \mathit{lfp}(\mathcal{R}(C)) = \emptyset$ implies $\mathit{TNI}(C,\mathsf{Z},f)$. Practically, this means that we can prove $\mathit{TNI}(C,\mathsf{Z},f)$ by computing the least fixpoint over the CHCs $\mathcal{R}(C)$ (e.g., using a datalog engine) and then check whether

it contains any fact from $\mathsf{P}^C_{\mathsf{Z},f}$. For components in Z, we assume a function *toVar* that maps components of the EVM semantic domain to CFG variables. The dependency predicates constituting a security pattern for trace noninterference are defined as

$$\mathsf{P}^C_{\mathsf{Z},f} := \{\mathit{InstMayDepOn}(pc, \mathit{toVar}(\mathsf{z})) \mid \mathsf{z} \in \mathsf{Z} \ \wedge \ C(pc) = op(\vec{x}, pc_{next}, pre) \ \wedge \ f(op)\} \cup$$
$$\{\mathit{VarMayDepOn}(x_i, \mathit{toVar}(\mathsf{z})) \mid \mathsf{z} \in \mathsf{Z} \wedge pc \in \mathsf{dom}(C) \wedge C(pc) = (op(\vec{x}, pc_{next}, pre)) \wedge f(op) \wedge x_i \in \vec{x}\}.$$

Theorem 3 (Soundness of trace noninterference)**.** *Let C be a contract, Z a set of components, and f an instruction-of-interest predicate. Then it holds that*

$$(\forall p \in \mathsf{P}^C_{\mathsf{Z},f}.\ p \notin \mathit{lfp}(\mathcal{R}(C))) \Rightarrow \mathit{TNI}(C, \mathsf{Z}, f).$$

Theorem 3 shows that $\mathsf{P}^C_{\mathsf{Z},f}$ is a security pattern for trace noninterference. The absence of facts from $\mathsf{P}^C_{\mathsf{Z},f}$ in $\mathit{lfp}(\mathcal{R}(C))$ ensures that the reachability of all instructions satisfying f is independent of variables representing components in Z and that all arguments x_i of such instructions are independent of z as well. These independences imply trace noninterference since they ensure that in two executions starting in configurations equal up to Z, all instructions satisfying f are executed in the same order (otherwise their reachability would depend on Z) and with the same arguments (otherwise their argument variables would depend on Z). Consequently, such executions produce the same traces, when only considering instructions satisfying f. A full proof of Theorem 3 can be found in [14].

5.5 Discussion

In this section, we presented a sound analysis pipeline for checking security properties for linearized EVM bytecode contracts by means of reasoning about dependencies between variables or instructions. While our work was inspired by Securify [26], we developed new formal foundations for the dependency analysis of EVM bytecode contracts and in this way revealed several sources of unsoundness in the analysis of Securify. Further, we provide soundness proofs for the analysis pipeline end-to-end; all theorems and proofs are available in the extended version of this paper [14]. The key pillars of the soundness proof are i) that our EVM CFG semantics satisfies all conditions to be used with the slicing framework [27], ii) that the EVM linearized bytecode semantics and the CFG semantics are equivalent, iii) that our set of CHCs encodes an over-approximation of dependencies in an EVM contract, and iv) that the generic security pattern $\mathsf{P}^C_{\mathsf{Z},f}$ is a sound approximation of trace noninterference. The proofs are valid under assumptions that are clearly stated in this paper.

We assume that EVM smart contracts are provided in a (stack-less) linearized form. Transforming into such a representation from a stack-based one is a well-studied problem [18] and a standard step performed by most static analysis tools [8, 26]. Up to this requirement, our analysis is parametric with respect to other preprocessing steps. More precisely, our analysis pipeline is sound for contracts with sound preprocessing information, and hence, in particular, for contracts without any preprocessing information but jump destinations needed for the CFG (cf. Section 5.1). This gives the flexibility, to enhance the precision of the analysis through the incorporation of soundly precomputed values — works on soundly precomputing jump destinations for EVM bytecode already exists [9].

6 Evaluation

The focus of this paper is on the theoretical foundations of a sound dependency analysis of smart contracts. However, we demonstrate the practicality of the presented approach by developing

contracts	errors	timeouts	contracts \(errors ∪ timeouts)	∅ time (ms)
720	**H** 34 **S** 34	**H** 46 **S** 30	634	**H** 7055 **S** 3107

Figure 4: Large-scale evaluation of HoRStify (**H**) and Securify (**S**).

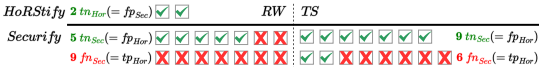

Figure 5: Classification of mismatching results of HoRStify and Securify for the RW (left) and TS (right) property. [3] [4]

the prototype analyzer HoRStify. We do not implement the logical rules from Section 5.3 directly in Soufflé (as done by Securify), but encode them in the HoRSt specification language [23]. The HoRSt language is a high-level language for the specification of CHCs. By introducing this additional abstraction layer, we get a close correspondence between our theoretical rules and their actual implementation and, hence, anticipate a lower risk of implementation mistakes that may invalidate soundness claims in the implementation.

HoRStify accepts as input a set of dependency facts encoding the security patterns specified in the HoRSt language and Ethereum smart contracts in the EVM bytecode format. It first invokes Securify's decompiler to transform the contract into a linearized representation and does some lightweight preprocessing to obtain the precomputable values (cf. Section 5.1). Then, HoRStify uses our formal specification of the CFG construction rules and the HoRSt framework to create a Soufflé executable for the analysis and invokes it.

To reduce the risks of implementation mistakes, we conduct a large-scale evaluation of HoRStify and Securify on real-world contracts. To this end, we use the sanitized dataset from [23] that consists of 720 distinct smart contracts from the Ethereum blockchain. We compare the performance of Securify and HoRStify on this dataset for both the RW pattern and for timestamp independence (TS) as defined in Section 5.4. We manually inspect all contracts on which Securify and HoRStify report a different result.

Figure 4 shows the evaluation results. The average execution time of HoRStify is approximately 2.3 times longer than for Securify. Consequently, HoRStify suffers from more timeouts than Securify; the execution of both tools is aborted after one minute. Figure 5 visualizes the manual classification for those smart contracts where HoRStify and Securify disagree. There are only two contracts where HoRStify matches the corresponding pattern, but Securify does not. Recall that for a sound tool, a pattern match indicates the discovery of provable independencies that imply either property violation (RW) or compliance (TS). An erroneous pattern match by HoRStify would present a soundness issue (false negative). We carefully examined the two examples and could confirm them not to constitute false negatives of HoRStify but false positives of Securify (fp_{Sec}), unveiling an imprecision of Securify. This seems surprising since our analysis generally tracks more dependencies than the one of Securify. However, while HoRStify implements standard control dependence to encode control dependencies (e.g., to compute join points after loops), Securify implements a less precise custom algorithm.

The contracts where Securify matches a pattern, but HoRStify does not, can either reveal soundness issues (false negatives) of Securify (fn_{Sec}) or a precision loss (false positives) of HoRStify (fp_{Hor}). Indeed, in the 29 contracts that are flagged only by Securify, we find both cases (as shown at the bottom of Figure 5), as we have evaluated in [14].

Overall, based on our evaluation results, we can bound the precision loss of HoRStify

[3] Ticks indicate correct matches (tn) and crosses wrong matches (fn) of the respective tool. tn_{Hor}/tn_{Sec}, fn_{Hor}/fn_{Sec}, tp_{Hor}/tp_{Sec}, fp_{Hor}/fp_{Sec} denote true negatives, false negatives, true positives, and true negatives of HoRStify/Securify, respectively.

[4] For TS we only consider the 165 contracts from the dataset containing a TIMESTAMP opcode, as Securify labels other contracts as trivially secure. The manual classification is a conservative best-effort estimate.

w.r.t. Securify. More concretely, when considering that Securify has a specificity[5] of S_{Sec} on the full dataset, then one can easily show that it holds for the specificity S_{Hor} of HoRStify that $\mathsf{S}_{Hor} \geq \mathsf{S}_{Sec} + \frac{tn_{Hor} - tn_{Sec}}{|dataset|}$ where tn_{Hor} are the true negatives for HoRStify, and tn_{Sec} are the true negatives for Securify found within the manually inspected mismatching contracts. Inserting the results from Figure 5, we can show that S_{Hor} can be at most 0.5 percentage points less than S_{Sec} for RW on the given dataset and at most 5.4 percent points less for TS.

7 Related Work

Most static analyzers are bug-finding tools (such as Oyente [21], EthBMC [7], and Maian [22]) that aim to reduce the number of contracts that are wrongly claimed to be buggy (false positives). To this end, these tools usually rely on the symbolic execution of the contract under analysis. The only example of a tool, which comes with a provable soundness claim, so far, is the analyzer eThor [23], whose analysis relies on abstract interpretation.

Symbolic execution and abstract interpretation have in common to target *reachability properties*. However, many generic security properties for smart contracts (as defined in [11]) fall into the broader category of *2-safety properties*. To check 2-safety properties with tools whose analysis is limited to reachability properties (such as eThor) requires an overapproximation of the original property in terms of reachability. In [11], it is, e.g., shown how to overapproximate the call integrity 2-safety property (characterizing the absence of reentrancy attacks) by a reachability property (single-entrancy) and two other properties, which are captured by our notion of trace noninterference. HoRStify (inspired by the unsound Securify tool [26]) devises a different analysis technique, which immediately accommodates the analysis of trace noninterference. As opposed to the analysis underlying eThor, this technique does not allow for verifying general reachability properties, but a special class of 2-safety properties (including trace noninterference). HoRStify and eThor, hence, can be seen as complementing tools that target incomparable property classes. The call integrity property falls neither in the scope of eThor nor HoRStify, but its overapproximation decomposes it into trace noninterference properties and a reachability property. Other generic security properties from [11] for characterizing the independence of miner-controlled parameters (including timestamp independence) immediately constitute trace noninterference properties and as such can be analyzed by HoRStify but not by eThor. More complex properties involving both universal and existential quantification of execution traces [5, 6] cannot be checked by either HoRStify or eThor.

8 Conclusion

In this work, we present the first provably sound static dependency analysis for EVM bytecode. Taking up the approach of the state-of-the-art static analyzer Securify [26], we replace the underlying analysis and spell out formal soundness guarantees. We can show that the resulting analysis is flexible enough to soundly characterize a generic class of non-interference-style properties, such as timestamp independence. We demonstrate the practicality of the approach by providing the prototypical analyzer HoRStify. We show that it can verify real-world smart contracts, and even though being provable sound, shows performance comparable to Securify.

[5] The specificity is a standard precision measure and is calculated as $\frac{tn}{tn+fp}$

References

[1] Solidity programming language. https://soliditylang.org/, https://github.com/ethereum/solidity. Accessed: 2022-02-05.

[2] Filip Adamik and Sokol Kosta. Smartexchange: Decentralised trustless cryptocurrency exchange. In *International Conference on Business Information Systems*, pages 356–367. Springer, 2018.

[3] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Roman-Diez, and Albert Rubio. Don't run on fumes—parametric gas bounds for smart contracts. *Journal of Systems and Software*, 176:110923, 2021.

[4] Moritz Andresen. The biggest smart contract hacks in history or how to endanger up to us $2.2 billion. https://medium.com/solidified/the-biggest-smart-contract-hacks-in-history-or-how-to-endanger-up-to-us-2-2-billion-d5a72961d15d, 2016.

[5] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *CSF'08*, pages 51–65, 2008.

[6] Pedro R. D'Argenio, Gilles Barthe, Sebastian Biewer, Bernd Finkbeiner, and Holger Hermanns. Is your software on dope? - formal analysis of surreptitiously "enhanced" programs. pages 83–110. Springer, 2017.

[7] Joel Frank, Cornelius Aschermann, and Thorsten Holz. Ethbmc: A bounded model checker for smart contracts. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2757–2774, 2020.

[8] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186. IEEE, 2019.

[9] Ilya Grishchenko. *Static Analysis of Low-Level Code*. PhD thesis, Technische Universität Wien, 2020.

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

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

[12] Samer Hassan and Primavera De Filippi. Decentralized autonomous organization. *Internet Policy Review*, 10(2):1–10, 2021.

[13] Sebastian Holler, Sebastian Biewer, and Clara Schneidewind. HoRStify: Sound security analysis of smart contracts. In R. Piskac and A. Voronkov, editors, *Short Papers of the 24th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2023.

[14] Sebastian Holler, Sebastian Biewer, and Clara Schneidewind. HoRStify: Sound security analysis of smart contracts. https://arxiv.org/abs/2301.13769, 2023. Extended version.

[15] Vanita Jain, Akanshu Raj, Abhishek Tanwar, Mridul Khurana, and Achin Jain. Coin drop—a decentralised exchange platform. In *Cyber Security and Digital Forensics*, pages 391–399. Springer, 2022.

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

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

[18] Allen Leung and Lal George. Static single assignment form for machine code. *ACM SIGPLAN Notices*, 34(5):204–214, 1999.

[19] Yinsheng Li, Xu Liang, Xiao Zhu, and Bin Wu. A blockchain-based autonomous credit system. In *2018 IEEE 15th International Conference on e-Business Engineering (ICEBE)*, pages 178–186. IEEE, 2018.

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

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

[22] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference*, pages 653–663, 2018.

[23] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. ethor: Practical and provably sound static analysis of ethereum smart contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 621–640, 2020.

[24] Clara Schneidewind, Markus Scherer, and Matteo Maffei. The good, the bad and the ugly: Pitfalls and best practices in automated sound static analysis of ethereum smart contracts. In *International Symposium on Leveraging Applications of Formal Methods*, pages 212–231. Springer, 2020.

[25] Alexandra Sims. Decentralised autonomous organisations: Governance, dispute resolution and regulation. *Dispute Resolution and Regulation (May 31, 2021)*, 2021.

[26] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018.

[27] Daniel Wasserrab, Denis Lohner, and Gregor Snelting. On pdg-based noninterference and its modular proof. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 31–44, 2009.

[28] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[29] Lanfranco Zanzi, Antonio Albanese, Vincenzo Sciancalepore, and Xavier Costa-Pérez. Nsbchain: a secure blockchain framework for network slicing brokerage. In *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2020.

[30] Michal Zima. Coincer: Decentralised trustless platform for exchanging decentralised cryptocurrencies. In *International Conference on Network and System Security*, pages 672–682. Springer, 2017.