HoRStify: Sound Security Analysis of Smart Contracts

Sebastian Holler‡*, Sebastian Biewer†, Clara Schneidewind*
*Max-Planck-Institute for Security & Privacy
Universitätsstraße, Bochum, Germany
†Saarland University, ‡Saarbrücken Graduate School of Computer Science
Saarland Informatics Campus, Saarbrücken, Germany

***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.

***Index Terms*—Ethereum, Smart Contract, Blockchain, Dependency Analysis, Security, Tool**

I. 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 [30], 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 [20], [31], decentralized exchanges [2], [16], [32] or decentralized autonomous organizations [14], [26]. 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 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.

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.

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 [24], [25], 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 [27], ZEUS [18], EtherTrust [12], NeuCheck [21], and eThor [24]. The soundness claims of ZEUS, Securify, EtherTrust, and NeuCheck are systematically confuted in [25] and [24].

The analysis tool eThor [24] 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. [13] 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 [27]—which was empirically shown unsound in [24].

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 [29], 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 [13]. Concretely, we make the following contributions:

- We study the root causes of the soundness issues of the state-of-the-art Ethereum smart contract analysis tool Securify [27] that so far had only been reported through empirical evidence in [24], [25]. In this course, we uncover new soundness problems in Securify's analysis, which we can show to affect real-world smart contracts.
- We devise a new dependency analysis for EVM bytecode based on program slicing following the static program framework presented in [29].
- We prove this dependency analysis to be sound with respect to a formal semantics of EVM bytecode.
- We show how to approach relevant smart contract security properties presented in [13] with the dependency analysis.
- We present HoRStify, an automated prototype static analysis tool that implements the dependency analysis.
- 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 II overviews our approach; Section III introduces the necessary background on Ethereum smart contract execution; Section IV discusses the challenges in designing sound static analysis tools for smart contract dependency analysis; Section V introduces the slicing proof framework from [29] that our analysis builds on; Section VI presents our static analysis based on program slicing and its soundness proof; Section VII reports on our prototype implementation HoRStify and its practical evaluation; and Section IX concludes the paper.

II. 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 [29] 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

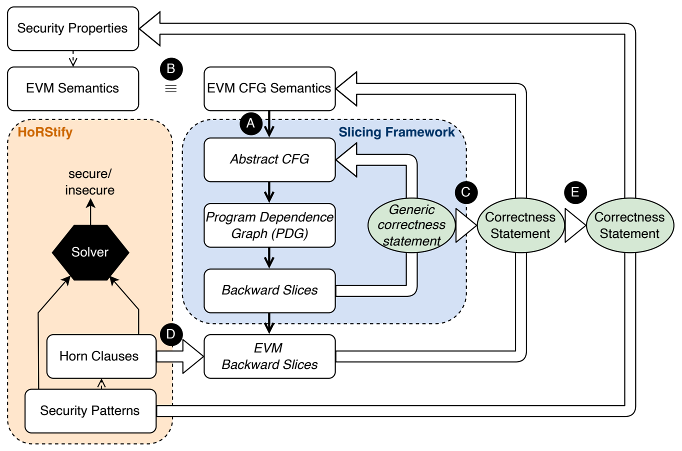


Fig. 1. Overview on the formal guarantees of HoRStify

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 [27] 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 will show in Section IV, 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 [29] 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 [15] for the technical details.

III. Background on Ethereum Smart Contracts

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 and the resulting particularities of the Ethereum smart contract execution environment.

a) 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.

b) 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. We will introduce relevant Solidity language features throughout the paper when needed.

c) 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. The following example illustrates how this peculiarity can be exploited in a smart contract:

```
function spinWheel() private (uint) {
        return block.timestamp % 37;  }
```

The function `spinWheel()` implements a spinning wheel that determines a random number between 0 and 36 based on the block timestamp (accessed via `block.timestamp`). Based on such a function, a contract could implement a roulette game where players bet money on the outcome of the spinning wheel. While the system timestamp may serve as a decent source of randomness for programs that run locally, this is not the case for smart contracts. A miner could easily tweak the timestamp of a block containing an invocation of the `spinWheel` function and thereby influence its outcome. In this way, a miner could ensure to win the roulette game themself.

IV. Challenges in Sound Dependency Analysis

As recently demonstrated in the literature [25], the sound analysis of Ethereum smart contracts is a challenging problem; most analysis tools aiming at provable soundness guarantees fall short of their goal. This can be mainly attributed to the non-standard language features of the EVM bytecode language and the unusual execution model of the EVM: Smart contracts

$$\begin{aligned} \textit{compliance: } & \textit{all } \mathrm{jump}(L_1, Y, \_), \mathrm{sstore}(L_2, \_, \_). \\ & \quad \textit{MustFollow}(L_1, L_2) \wedge \textit{DetBy}(L_1, Y, \mathrm{caller}) \\ \textit{violation: } & \textit{some } \mathrm{sstore}(L_1, X, \_). \\ & \quad \neg\textit{MayDepOn}(X, \mathrm{caller}) \wedge \neg\textit{MayDepOn}(L_1, \mathrm{caller}) \end{aligned}$$

Fig. 2. Restricted Write compliance and violation pattern [27]

are executed in a (potentially) hostile environment, which can interact with, and even, schedule contracts. The smart contract execution is dependent on the gas resource and the low-level EVM bytecode language features little static information. As a consequence, execution heavily depends on unknown runtime parameters, which makes it hard to reason statically about contract behaviors in a sound and reasonably precise and efficient manner. This incentives the incorporation of ad-hoc optimizations, which increase the complexity of the analysis even further. Consequently, it is crucial to establish rigorous formal foundations for EVM bytecode analysis and to align the implementation with these foundations. In the following, we demonstrate how the lack of formal foundations affects the guarantees of the state-of-the-art analysis tool Securify [27].

A. 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 [17]. A *compliance pattern is sound w.r.t. a property*, if satisfying the pattern implies satisfaction of the property, and, analogously, a *violation pattern is sound w.r.t. a property* if satisfying the pattern implies violation of the property. If neither of the patterns is satisfied, the satisfaction of the property is inconclusive. Obviously, it cannot be that for the same contract and for the same property a sound compliance and violation pattern hold simultaneously.

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. Figure 2 shows a compliance[1] and violation pattern for RW [27].

The compliance pattern for RW states that for all conditional jump instructions at program location L_1 that branch on condition Y ($\mathrm{jump}(L_1, Y, \_)$) and for all storage write instructions

[1]The Securify implementation contains two compliance patterns; one is shown in [27], the other one is shown in Figure 2.

```
contract Start { bool test = false;
    function flipper() public {
        if (uint(msg.sender) * 0 == 0)
        {   test = !test; } } }
```

Fig. 3. Securify counterexample: must-analysis

at location L_2 ($\mathrm{sstore}(L_2, \_, \_)$) that are necessarily preceded by such jump instructions ($\textit{MustFollow}(L_1, L_2)$), it must hold that at location L_1 the condition Y must be determined by the caller of the transaction ($\textit{DetBy}(L_1, Y, \mathrm{caller})$). The violation pattern for RW states that there is some storage write instruction at location L_1 writing to storage address X ($\mathrm{sstore}(L_1, X, \_)$) such that that neither the address X nor the execution of the storage instruction at L_1 may depend on the caller of the transaction ($\neg\textit{MayDepOn}(X, \mathrm{caller}) \wedge \neg\textit{MayDepOn}(L_1, \mathrm{caller})$).

B. Soundness Issues

Even though Securify characterizes security properties and their corresponding compliance and violation patterns, no formal connection between patterns and properties is established. In particular, they do not prove the soundness of the patterns they propose in [27] w.r.t. the properties they are supposed to approximate. Doing so would require 1) to prove that the dependency predicates imply semantic notions of independence (sound core analysis) and 2) to prove that the semantic notions implied by the security patterns indeed imply the security properties (sound security patterns). In the following, we use the example of the RW property to show how the absence of formal soundness arguments causes Securify to miss corner cases that undermine its soundness guarantees.

1) Sound Core Analysis: Securify does not draw a connection between the dependency predicates and the EVM bytecode semantics. This leads to mismatches between the intuitions for the predicates and their definitions.

a) Must-analysis: Securify's dependency analysis and predicates can be attributed to one of two categories: a *may-analysis* aims at over-approximating possible control and dataflow dependencies, encoded by *may-predicates*, and a *must-analysis* aims at capturing dependencies and deducing *must-predicates* that show a definite effect on the actual execution. According to their usage in the security patterns, negated may-predicates imply a notion of independence, while must-predicates should imply a form of determination. More precisely, it is stated that the must-predicate $\textit{DetBy}(L, Y, T)$ "indicates that a different value of T guarantees that the value of Y changes." [27] This guarantee, however, is violated in the contract shown in Figure 3, where Securify inferred that `test` is determined by the caller although every caller can change the value of `test`: The check of the conditional evaluates to `true` for any value of `msg.sender`, hence allowing every caller to write the `test` field. Still, Securify reports this contract to match the compliance pattern, indicating that the condition in line 3 would be determined by the caller. The underlying reason for this problem is of substantial nature: The must-analysis under-

```
1  contract Start { bool test = false;
2      function storeTest(uint c) public {
3          address[] memory a = new address[](7);
4          for (uint i = 0; i < 7; i++) {
5              a[i] = msg.sender;}
6          if (a[0] != address(0)) {test = !test;} } }
```

Fig. 4. Securify counterexample: storage abstraction

```
1  contract Start { bool test = false;
2      address a;
3      function setAddress(address addr) public
4      {   a = addr;   }
5      function flipper () public {
6          try Start(this).setAddress(msg.sender) {
7              if (a != address(0)) { test = !test; }
8          } catch { revert();  }  }  }
```

Fig. 5. Securify counterexample: reentrancy handling

```
contract Check {
        function testZero (address a) public {
            assert (a == address(0)); } }
contract Start {
        bool test = false;
        address check = address(42);
        function flip() public {
            try Check(check).testZero(msg.sender){
                test = !test;
            } catch {return;} } }
```

Fig. 6. Securify counter example: external call handling

approximates control flows but over-approximates data flows. More precisely, a variable X is considered to be determined by a variable Y if Y occurs in the expression assigned to X. Since `msg.sender` appears in the condition expression in line 3, the condition is considered to be determined by `msg.sender` even though it actually is independent of `msg.sender`. This treatment makes the must-analysis inherently unsound.

Due to this substantial mismatch between the intuition for the *DetBy* predicate and its implementation, it is unclear whether adjusting the implementation of the must-analysis such that it is sound, could result in a performant and precise analysis. So, in this work, we will focus on the may-analysis.

b) Memory Abstraction: For establishing a sound may-analysis, it is crucial to overapproximate dependencies for all relevant system components that can interact with each other. In particular, this includes stack, memory and storage variables, because values are written from the stack to the local memory and persistent storage, and back. However, the addresses of memory and storage accesses are not statically known but specified on the stack. E.g., the EVM instruction $\mathsf{MSTORE}(x, y)$ denotes that the value in stack variable y should be written to the address as given in stack variable x. Consequently, the concrete memory address at which the value in y will be stored may only be known at runtime. This poses a big challenge to static analysis since for precisely modeling the dependencies on different memory and storage cells, their accesses need to be known statically. Otherwise, the dependencies on all memory and storage cells would need to be merged, resulting in a substantial precision loss. In practice, memory and storage addresses can in most cases be precomputed by partial evaluation [27]. Hence, this preprocessing information can be used to enhance the analysis precision.

Securify implements this optimization in an unsound way, as illustrated by the example in Figure 4. Here, function `storeTest` locally defines a new address array `a` of size 7 and initializes all its elements with the contract caller `msg.sender`. The write access to the `test` variable is restricted by the condition that the first array element `a[0]` (which obviously contains the caller address) is not 0. Consequently, the contract satisfies the RW property. Still, Securify certifies a violation of the RW pattern w.r.t. `test`[2]. The example illustrates that the analysis does not consider that a memory address may be statically unknown at the point of writing but known at the point of reading. Since writing to the array is done in a loop, for the assignment `a[i] = msg.sender` the memory address cannot be statically determined. For the condition in line 6, in contrast, the memory address for `a[0]` can be precomputed. However, Securify fails to account for the fact that dependencies of an unknown memory access should propagate to all concrete memory addresses.

c) Reentrancy handling: Smart contracts are reactive programs in the sense that they can transfer control to other contracts and are subject to *reentrancy*, i.e., while awaiting the return of the other contract, this contract may call the waiting contract again. Figure 5 shows a simple case of reentrancy. In this variant of Figure 3, function `flipper` calls the contract's function `setAddress` within a new internal transaction[3]. `flipper` uses `setAddress` to store the caller `msg.sender` in the storage location `a` (defined in line 2). Then, `flipper` modifies the critical storage location `test` if and only if the address stored in `a` is not zero. Ethereum contracts are executed non-concurrently, so the value of `a` remains unchanged after line 6 and before the evaluation of the condition in line 7.

Consequently, a caller with address 0 can never write to the `test` field and the contract satisfies the RW property w.r.t. `test`. Still, Securify reports a match of the violation pattern for `test`. Inspection of the Securify code reveals that it does not model potential dependencies between arguments of external calls and storage locations accessible via reentrancy.

d) External call handling: Aside from reentrancy, external calls may affect the local execution state in multiple ways.

The success of an external call is indicated by placing a corresponding flag on the stack and the return value of the call (if existent) is written to a memory fragment that is specified as an argument to the call. These effects may depend on the recipient and the arguments of the call. The example in Figure 6 illustrates how ignoring those dependencies causes an unsoundness in Securify: In this example, the sender check is outsourced to the method `testZero` of another contract. The assignment of variable `test` depends on whether `testZero`

[2]This is indeed unsoundness and not imprecision: Securify guarantees that a property does not hold if the violation pattern matches. Only inconclusive cases (i.e., no compliance and no violation pattern matches) cause imprecision.

[3]A reasonable contract would call a function of the same contract directly so that such a call would be translated to a JUMP by the compiler. The chosen syntax enforces that the function call will be translated to a CALL instead.

```
1  contract Start { bool test = false;
2      function flipper () public {
3          require(gasleft() > 10000);
4          bool flip = false;
5          if (msg.sender == address(0)) {
6          {   while (gasleft() >= 5000)
7                  { flip = !flip; } }
8          if (gasleft() < 5000) {test = flip;} } }
```

Fig. 7. Securify counterexample: gas handling

returns without the `assert` throwing an exception, which in turn depends on (the input data) `msg.sender`. Hence, this contract satisfies the RW property. Still, Securify reports a violation, since no dependencies between the input to the call and the call output are modeled.

e) Gas handling: Figure 7 shows a contract that indirectly restricts write access to storage `test` by consuming the gas resource in a controlled way. In line 3, the contract ensures that it is executed with a generous amount of gas; if not enough gas is available, the execution is aborted and no caller is able to write to `test`. The code between lines 5 and 7 essentially wastes masses of gas if the caller address is equal to 0, and, otherwise, consumes very little gas. The crux of the contract is in line 8: From the amount of gas that is left, the contract can determine if the caller's address is equal to 0—this is the case if and only if less than 5000 gas units are left. Hence, depending on the amount of available gas, either no caller or only caller 0 can write to `test`. So, there is always at least one caller that cannot write to `test`—the contract satisfies the RW property. However, Securify reports a violation of this property. The reason for this wrong analysis result is that Securify does not track dependencies for the gas resource.

2) Sound Security Properties: Since the dependency predicates do not have a semantic characterization, the soundness of the security patterns w.r.t. their corresponding property cannot be proven. Indeed, Schneidewind et al. [24] provide counter examples for the soundness of 13 out of the 17 security patterns given in [27]. Above that, the unsoundness of the RW property undeniably manifests in line 4 of the contract we constructed in Figure 5. For this example, Securify reports simultaneously(!) satisfaction of a compliance and a violation pattern for the RW property w.r.t. `a`. This refutes the claim that compliance and violation patterns constitute sufficient criteria for property compliance and violation, respectively.

V. Analysis Foundations

To design a sound static analysis for EVM bytecode based on program slicing, we instantiate the slicing proof framework from [29] with a formal bytecode semantics as defined in [13]. Before discussing the instantiation in Section VI, we shortly overview both frameworks.

A. EVM bytecode semantics

The EVM semantics was formally defined in [13] 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 $\Gamma \vDash S \to S'$. 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 $\Gamma \vDash S \to^* S'$ 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 [13]. 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).

a) 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.

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

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'.

b) Security properties: Previous work [13] 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 [13] 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, [13] 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 [13] 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*

$$\Gamma =_{/\mathit{timestamp}} \Gamma' \;\wedge \Gamma \vDash s_C :: S \xrightarrow{\pi}{}^{*} s'_C :: S \;\wedge\; \mathit{final}\,(s')$$

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

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 $\Gamma =_{/\mathit{timestamp}} \Gamma'$) 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.

B. 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. [29], which establishes a language-independent correctness result for slicing based on abstract control flow graphs (CFGs).

a) 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 $\langle n,\ \theta\rangle - a \rightarrow \langle n',\ \theta'\rangle$.

b) 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 $\mathit{BS}(n)$) is defined as the set of all nodes n' that can reach n within the PDG.

c) Correctness statement: The generic correctness statement for slicing proven in [29] is stated as follows:

Theorem 1. *Correctness of Slicing Based on Paths [29]*

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

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' ($\langle n,\ \theta\rangle \xrightarrow{as'}{}^{*}_{\mathit{BS}(n')} \langle n',\ \theta''\rangle$) without altering the path through the PDG in any other way ($as \downarrow_{\mathit{BS}(n')} = as'$) 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 ($\forall V \in \mathit{Use}(n').\ \theta'(V) = \theta''(V)$). We call the PDG without the above-mentioned edges also *sliced PDG* or *sliced graph.*

VI. Sound EVM Dependency Analysis

In the following, we instantiate the slicing proof framework [29] 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 [13].

A. 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 $\bot$ 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. Environmental information that cannot be directly accessed during the execution (such as the storage of other contracts) is hidden in the dedicated global environmental variable $\mathsf{external}^{eg}$.

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).

$$\frac{C(pc) = (\mathsf{JUMPI}({x_1}^{ls}, {x_2}^{ls}), pc_{\text{next}}, pre) \qquad f \;=\; (\lambda\theta.\theta \leftarrow g^e := \theta[g^e] - 10)}{\mathcal{C}, cd \vDash \; (pc, 0) - \Uparrow f \rightarrow (pc, 1)}$$

$$Def \;=\; \{g^e\} \qquad Use \;=\; \{g^e\}$$

$$\frac{C(pc) = (\mathsf{JUMPI}({x_1}^{ls}, {x_2}^{ls}), pc_{\text{next}}, pre) \qquad Q \;=\; (\lambda\theta.\theta[{x_2}^{ls}] = 0)}{\mathcal{C}, cd \vDash \; (pc, 1) \; -(Q)_{\surd} \rightarrow (pc_{\text{next}}, 0)}$$

$$Def \;=\; \emptyset \qquad Use \;=\; \{{x_2}^{ls}\}$$

Fig. 8. JUMPI abstract CFG instantiation

Figure 8 shows two exemplary rules for the conditional jump instruction JUMPI. The first argument to JUMPI is the jump destination and the second argument is the condition variable that must be non-zero for the jump to happen. We only show rules for the case that the condition is false, i.e., the jump does not happen. The upper rule defines a state-changing edge that deducts the gas that has to be paid for a JUMPI instruction. Appropriately, both Def and Use sets contain the gas variable because the current gas value must be read from and the reduced value updated in state θ. Note that the edge goes from the initial node for *pc* to an intermediate node for *pc*, because a second step is necessary to decide whether the program should jump. The second step, depicted by the lower rule, continues in the intermediate node for *pc* and checks if the condition (in variable ${x_2}^{ls}$) is false (i.e., if it is zero) via a predicate edge. In this case, the execution proceeds to the initial node representing pc_{next}. ${x_2}^{ls}$ is the only variable used by Q, hence it is the only variable in the Use set.

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.

B. Core abstractions

We review the most interesting aspects of the CFG semantics and how they lead to a precise dependency analysis. In this course, we will show how to overcome the challenges presented in Section IV.

a) Gas abstraction: In the EVM, the execution of instructions consumes gas. If the gas is not sufficient to finish the execution of a contract, it is aborted with an exception. Modeling this behavior accurately would result in a very imprecise analysis, since, technically, every instruction would be control-dependant on all its preceding instructions. This is as the execution of an instruction depends on whether prior instructions led to an out-of-gas exception. However, in practice, users should only call contracts with a sufficient amount of gas since, otherwise, the contract execution exceptionally halts. For this reason, there exist static analysis tools for computing

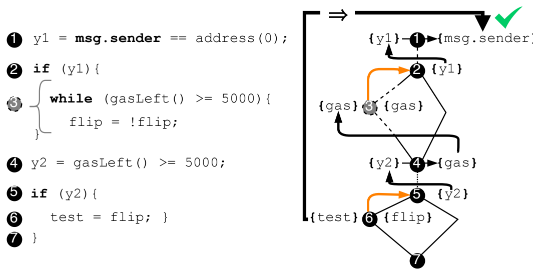
$$\frac{C(pc) = (\mathsf{ADD}(y^{ls}, {x_1}^{ls}, {x_2}^{ls}), pc_{\text{next}}, pre) \qquad f \;= (\lambda\theta.\theta \leftarrow g^e := \theta[g^e] - 3)}{\mathcal{C}, cd \vDash\; (pc, 0) - \Uparrow f \rightarrow (pc, 1)}$$

$$Def \;=\; \{g^e\} \qquad\qquad Use \;=\; \{g^e\}$$

$$\frac{\begin{array}{c} C(pc) = (\mathsf{ADD}(y^{ls}, {x_1}^{ls}, {x_2}^{ls}), pc_{\text{next}}, pre) \\ f \;= (\lambda\theta.\theta \leftarrow y^{ls} := \theta[{x_1}^{ls}] + \theta[{x_2}^{ls}]) \end{array}}{\mathcal{C}, cd \vDash\; (pc, 1) - \Uparrow f \rightarrow (pc_{next}, 0)}$$

$$Def \;=\; \{y^{ls}\} \qquad\qquad Use \;=\; \{{x_1}^{ls}, {x_2}^{ls}\}$$

Fig. 9. CFG semantics rules for the ADD instruction.



```
❶ y1 = msg.sender == address(0);
❷ if (y1){
③   { while (gasLeft() >= 5000){
        flip = !flip;
  }
❹ y2 = gasLeft() >= 5000;
❺ if (y2){
❻   test = flip; }
❼ }
```

Fig. 10. Example control flow with gas dependencies. Def sets are given at the left of each node, Use sets at the right. Data dependencies are indicated by black arrows, control dependencies by orange ones.

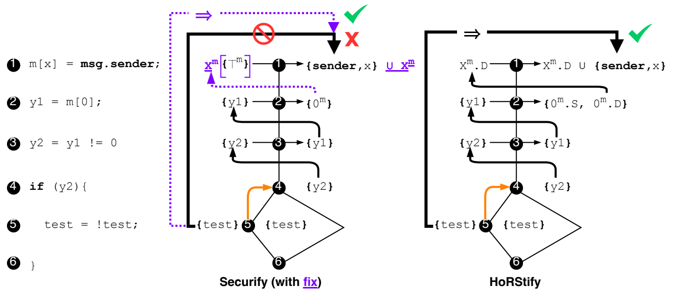
(sound) gas bounds [3] and even Solidity's online compiler provides gas estimates for smart contract execution.

Hence, for our analysis we assume that a contract does not run out of gas and do not model the corresponding behavior in the CFG semantics. We remark that Securify also makes this assumption implicitly; we spell it out explicitly as follows:

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.*

In contrast to Securify, we do not ignore gas entirely, but model the gas reduction for all instructions. This allows capturing dependencies such as the one highlighted in Figure 7 (and missed by Securify). In the CFG, we always model the gas reduction as a separate edge involving an intermediate node (e.g., with the upper rule in Figure 8). The Def set of one node contains only the gas variable, while the Def set of the other node only contains the (stack) variables involved in the actual instruction. An example for that is given by the (simplified) CFG rules of the ADD instruction in Figure 9. Technically, an ADD instruction performs two types of state updates: it decreases the gas and performs addition on stack variables. Since those two state updates are independent, their execution can be split into two different nodes. As a consequence, the node $(pc, 1)$ is not data-dependent on nodes writing the gas variable.

Still, the gas abstraction is sound (under Assumption 1) and correctly captures the dependencies of the example in Figure 7: Figure 10 shows an incomplete and simplified CFG of the example in Figure 7 with annotated Def and Use sets. The example illustrates how the CFG captures the dependency of the storage write (`test = flip`) on the **`msg.sender`** variable. The storage write in ❻ is control dependant on the conditional `y2` in ❺, and ❺ depends on node ❹ where `y2` is defined. ❹ accesses the gas value, so a dependency between ❹ and the gas nodes is established. Node ❸ is one of these gas nodes (there are more not shown in the picture). The execution of ❸ depends on condition `y1` checked in ❷, so it is control dependant on ❷. Node ❶ defines `y1`, so ❷ depends on ❶. Thus, there is a transitive dependency between writing to `test` in ❻ and reading **`msg.sender`** in ❶.

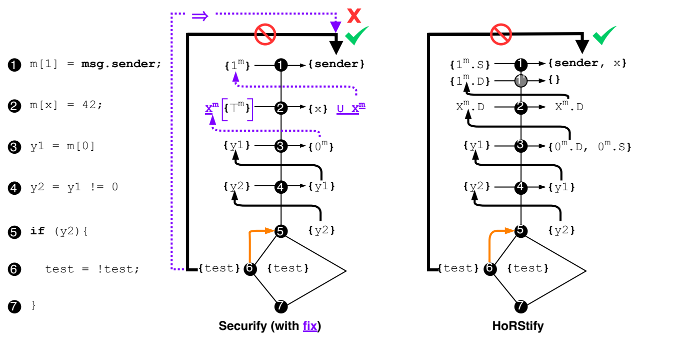


```
❶ m[x] = msg.sender;
❷ y1 = m[0];
❸ y2 = y1 != 0
❹ if (y2){
❺   test = !test;
❻ }
```

Fig. 11. Simplified version of contract in Figure 4 satisfying the RW property with PDGs depicting the dependencies modeled by Securify and HoRStify.

b) 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 demonstrated in Section IV, using preprocessed data may introduce unsoundness. 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 revisit Figure 4 in a simplified version that is depicted as a CFG in Figure 11. The black and solid line parts of the left CFG visualize how Securify misses the dependency 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 11. Now, the dependency of the read access in ❷ to the write operation in ❶ is naturally established. One should



```
❶ m[1] = msg.sender;
❷ m[x] = 42;
❸ y1 = m[0]
❹ y2 = y1 != 0
❺ if (y2){
❻   test = !test;
❼ }
```

Fig. 12. Contract violating the RW property with PDGs depicting the dependencies as modeled by Securify and HoRStify.

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 12. This variant of Figure 11 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 12). 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.S stores values that are assigned to x when the memory location for x is statically known, and *D-variable* x^m.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.D, unless the memory location for x is statically known, in which case x^m.S stores the value and x^m.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

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

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. *load* is used in the inference rules in Figure 13 that define the memory read and write operations for the CFG semantics. In these rules, we use $X^m.S$ for the set of all S-variables and $X^m.D$ for the set of all D-variables. The leftmost MSTORE rule is for the case that a value is written to a memory location that could not be statically inferred (i.e., $\textit{pre}[0] = \bot$). There, any of the memory variables from $X^m.D$ might be redefined, hence the Def set contains all variables in $X^m.D$. As discussed for the hypothetical fix of Securify, also the Use set needs to include $X^m.D$, because we must not interrupt potential dependencies for memory cells that are not changed by this MSTORE instruction. An example for this is node ❷ in Figure 12 (right CFG). The S-variables are not part of the Use set and hence not part of the value intermingling in ❷. This removes the imprecision that occurred in the proposed hypothetical fix above. Still, the MLOAD rules make sure that no dependencies to S-variables are missed by adding both D-variables and S-variables to the Use set. This way, the connection between the memory location and the stored value is preserved; ${x_1}^m$.D does not inherit any data dependencies from ${x_2}^m$.S for locations $x_1 \neq x_2$. An example for that is given in Figure 12, where memory location `0` does not inherit the dependency from memory location `1` written in ❶. This is thanks to the node splitting at ❶ that breaks the propagation of dependencies on precomputed locations to dynamic ones.

c) Call Abstraction: Contract calls in Ethereum trigger a multitude of possible (side) effects. When calling another account, the control flow is handed over to the code residing in this account. This code may initiate further internal transactions, e.g., perform money transfers or even reenter the calling contract before reporting back the result to the callee.

This behavior poses a big challenge to sound static analysis since all possible effects of interactions with other (potentially unknown) contracts need to be over-approximated. Securify avoids this challenge by sacrificing soundness and ignoring all data dependencies arising from external calls (including effects of reentrancy) as demonstrated by the examples in Figure 5 and Figure 6. In contrast, to give a sound and precise characterization of these dependencies, we first simplify the problem by restricting our analysis to a set of well-behaved smart contracts and then model the remaining dependencies in a fine-grained manner.

The class of smart contracts that we target are such contracts that cannot write storage variables in reentering executions. This restriction rules out race conditions on contract variables and as such is a highly-desirable property that can be easily achieved (e.g., by a strict local locking discipline). We call contracts satisfying this restriction *store unreachable*:

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

The contract in Figure 5 trivially violates store unreachability since the field `a` can be written in a reentering execution. This could be easily fixed by guarding each function with a lock that blocks reentering executions. Store unreachability is a local reachability property of the contract under analysis and

$$\frac{\begin{array}{c}C(pc) = (\mathsf{MLOAD}(y^{ls}, x^{ls})), pc_{next}, pre) \\ pre[1] = \bot \\ f \ = (\lambda\theta.\theta \leftarrow y^{ls} := load\ \theta\ (\theta[x^{ls}]))\end{array}}{\mathcal{C}, cd \vDash\ (pc, 0) - \Uparrow f \rightarrow (pc, 1)}$$

$$Def \ = \ \{y^{ls}\} \qquad Use \ = \ X^m.D \cup X^m.S \cup \{x^{ls}\}$$

$$\frac{\begin{array}{c}C(pc) = (\mathsf{MSTORE}({x_1}^{ls}, {x_2}^{ls}), pc_{next}, pre) \\ pre[0] = \bot \\ f \ = (\lambda\theta.\theta \leftarrow \theta[{x_1}^{ls}].\mathsf{D} := \theta[{x_2}^{ls}])\end{array}}{\mathcal{C}, cd \vDash\ (pc, 0) - \Uparrow f \rightarrow (pc, 1)}$$

$$Def \ = \ X^m.D \qquad Use \ = \ X^m.D \cup \{{x_1}^{ls}, {x_2}^{ls}\}$$

$$\frac{\begin{array}{c}C(pc) = (\mathsf{MLOAD}(y^{ls}, x^{ls})), pc_{next}, pre) \\ pre[1] = \lfloor x^m \rfloor \\ f \ = (\lambda\theta.\theta \leftarrow y^{ls} := load\ \theta\ x^m)\end{array}}{\mathcal{C}, cd \vDash\ (pc, 0) - \Uparrow f \rightarrow (pc, 1)}$$

$$Def \ = \ \{y^{ls}\} \qquad Use \ = \ \{x^m.\mathsf{S}, x^m.\mathsf{D}\}$$

$$\frac{\begin{array}{c}C(pc) = (\mathsf{MSTORE}({x_1}^{ls}, {x_2}^{ls}), pc_{next}, pre) \\ pre[1] = \lfloor x^m \rfloor \\ f \ = (\lambda\theta.\theta \leftarrow x^m.\mathsf{S} := \theta[{x_2}^{ls}])\end{array}}{\mathcal{C}, cd \vDash\ (pc, 0) - \Uparrow f \rightarrow (pc, 1)}$$

$$Def \ = \ \{x^m.\mathsf{S}\} \qquad Use \ = \ \{{x_2}^{ls}\}$$

$$\frac{\begin{array}{c}C(pc) = (\mathsf{MSTORE}({x_1}^{ls}, {x_2}^{ls}), pc_{next}, pre) \\ pre[0] = \lfloor x^m \rfloor \qquad f \ = (\lambda\theta.\theta \leftarrow x^m.\mathsf{D} := \bot)\end{array}}{\mathcal{C}, cd \vDash\ (pc, 1) - \Uparrow f \rightarrow (pc, 2)}$$

$$Def \ = \ \{x^m.\mathsf{D}\} \qquad Use \ = \ \emptyset$$

Fig. 13. MLOAD memory abstraction instantiation

$$\frac{\begin{array}{c}C(\mathsf{pc}) = (\mathsf{CALL}(y^{ls}, g^{ls}, to^{ls}, va^{ls}, io^{ls}, is^{ls}, oo^{ls}, os^{ls}), \mathsf{pc}', pre) \\ f_1 = \lambda\theta.\theta \leftarrow y^{ls} := applyCall(\theta, C, \mathsf{pc})[y^{ls}] \\ f_2 = \lambda\Theta.\theta \leftarrow external^{eg} := applyCall(\theta, C, \mathsf{pc})[external^{eg}] \\ f = \lambda\theta. f_2(f_1(\Theta))\end{array}}{\mathcal{C}, cd \vDash\ (\mathsf{pc}, 0) - \Uparrow f \rightarrow (\mathsf{pc}, 1)}$$

$$Def \ = \ \{y^{ls}, external^{eg}\}$$

$$Use \ = \ \{g^{ls}, to^{ls}, va^{ls}, io^{ls}, is^{ls}, oo^{ls}, os^{ls}, g^{el}, \mathsf{actor}^{el}\} \ \cup \ X^m \cup \ X^{eg} \ \cup \ X^g$$

Fig. 14. Simplified CFG rule for the CALL opcode

as such falls in the scope of the sound analysis tool eThor [24] and hence can be automatically verified.

Even when focussing on store unreachable contracts, the program dependencies induced by external calls are manifold and often subtle. Figure 14 shows one (slightly simplified) rule of the CFG semantics for external calls. As seen in the previous examples, node splitting is used to separate the dependencies of different variables. The rule displayed in Figure 14 gives one of the rules for setting a call's return value (written to the stack variable y^{ls}) and updating the external environment (represented by variable $\mathsf{external}^{eg}$) according to the call effects.

To obtain the updated CFG state after a call, the rule uses the function *applyCall*, which executes the internal transaction initiated by the CALL opcode[4]. The CFG state resulting from this execution is then used to describe the state updates (in the case of the given rule, the updates on the variables y^{ls} and $\mathsf{external}^{eg}$, as indicated by the Def set). Even though the whole CFG state θ is taken as an argument by *applyCall*, not all variables in θ can influence all aspects of the state after returning. The variables that indeed may affect y^{ls} and $\mathsf{external}^{eg}$ are given in the Use set. More precisely, the result of a call may still depend on the global state, so all global environmental variables (X^{eg}), as well as the global variables of the contract under analysis itself (X^g). Additionally, the execution of the called contract can be influenced by the parameters given to the call: The argument g^{ls} attributes to the amount of gas given to the call, to^{ls} gives the address of the recipient account and va^{ls} the amount of money transferred with the call. The arguments io^{ls} and is^{ls} specify the memory fraction (offset and size) from which input data to the call is read and oo^{ls} and os^{ls} correspondingly define the memory fraction where the call's result data will be written. In the given simplified rule, we consider that the concrete memory fragments could not be precomputed and hence all memory (X^m) could potentially be input data to the call. The Use set also contains the calling account (as given in actor^{el}), since this information is made accessible during a call. Finally, the Use set contains the amount of gas that is available at the point of calling (given by g^{el}) since this value may influence the amount of gas given to the call.

We want to highlight two forms of dependencies, which may erroneously be assumed to be ruled out by the assumption of store unreachability: First, the Use set explicitly contains the storage variables (X^g) of the contract under analysis, even though we assume this contract to be store unreachable and (by the semantics) its storage variables cannot be accessed by any other contract. Second, both the Def and the Use set contain the variable $\mathsf{external}^{eg}$ that represents the external environment (in particular the state of other contract accounts). This implies that the rule in Figure 14 explicitly models information to be stored and retrieved from contract accounts during an external call. In Figures 15 and 16, we illustrate the need for these dependencies by two examples.

The example in Figure 15 shows how dependencies on a storage variable are introduced by reading a contract variable during a reentering execution. Note that store unreachability only assures that reentering executions can not write contract variables, but does not prevent read accesses. The example gives another version of the `Test` contract, which performs the check of **`msg.sender`** in an indirect way: First, it writes **`msg.sender`** to the contract variable `sender`. To read the variable again, a `RetrieveSender` contract `rs` is used as a proxy: [5] The `Test` contract calls `RetrieveSender`'s `getTestSender` function (in line 15), which in turn reenters `Test` via its `getSender` function (in line 3) to obtain the value of `sender`. This value is finally

[4]We define *applyCall* using the EVM semantics and hence can infer Def and Use sets from the corresponding EVM semantics rules.

[5]Note that in Ethereum, a contract is identified by its address. In Solidity, the syntax `RetrieveSender rs = RetrieveSender (address(42))` means that the contract at address 42 is assumed to be (of the type) RetrieveSender and accessible via variable `rs`.

```
1  contract RetrieveSender {
2    function getTestSender() public returns (address) {
3      try Test(msg.sender).getSender() returns (address a) {
4        return a; }
5      catch {return address(0); }}}
6
7  contract Test {
8    bool test = false;
9    address sender;
10   RetrieveSender rs = RetrieveSender (address(42));
11   function getSender () public returns (address) {
12     return sender;}
13   function flip () public {
14     sender = msg.sender;
15     try rs.getTestSender() returns (address a)  {
16       if (a != address(0)){
17         test = !test;}}
18     catch {return; }}}
```

Fig. 15. Example: Reading storage variables during reentering execution.

```
1  contract SaveAddr {
2    address addr = address(0);
3    function set(address a) public {
4      addr = a; }
5    function get( ) public returns (address) {return addr; }}
6
7  contract Test {
8    bool test = false;
9    SaveAddr sa = SaveAddr (address(42));
10   function flip () public {
11     try sa.set(msg.sender) {
12       try c.get() returns (address a)  {
13         if (sa != address(0)){
14           test = !test; } }
15       catch {return; }}
16     catch {return; } } }
```

Fig. 16. Example: Propagating dependencies via an external contract account.

returned to contract `Test`. As a consequence, the return variable `a` in line 16 contains the value of `msg.sender`, and so the assignment of variable `test` is dependent on `msg.sender`. This dependency, however, can only be tracked when considering that the contract's own storage variables may influence the return value of an external call.

The example in Figure 16 shows how dependencies can be propagated via another contract account. Note that store unreachability is a contract-specific property that only ensures that the contract under analysis is not written in reentering executions. The assumption does not restrict the storage modification of other contracts. The version of the `Test` contract given in Figure 16 uses the contract `SaveAddr` to propagate the value of `msg.sender`. To this end, it first writes the value of `msg.sender` into the `addr` storage variable of the `SaveAddr` contract `sa` using the `set` function (in line 11). Afterwards, it retrieves the value back by accessing `c`'s storage via the `get` function (in line 12). Consequently, the return variable `a` contains the value of `msg.sender` in line 13 what makes the following write to `test` dependent on that value. This dependency can only be faithfully modeled when considering that an external call may change the state of other accounts, and may also be influenced by this state. This motivates why the $\mathsf{external}^{eg}$ variable needs to be included in both the Def and the Use set of the rule in Figure 14.

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

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. $n_x \in \mathit{BS}(n_{\mathit{if}})$ 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 VI-D 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)**.**

$$\forall x\ y.\ \mathit{VarMayDepOn}(y, x) \notin \mathit{lfp}(\mathcal{R}(C)) \Rightarrow y \perp x$$

with $y \perp x$ 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 $\textit{VarMayDepOn}(y, x)$ is not included in $\textit{lfp}(\mathcal{R}(C))$ 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 $\textit{InstMayDepOn}(i, x)$, we refer to [15].

D. 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 $\textit{TNI}(C, \mathsf{Z}, f)$) is defined as follows:*

$$\begin{aligned}\textit{TNI}(C, \mathsf{Z}, f) := \forall\ \Gamma\ \Gamma'\ s\ s'\ t\ t'\ \pi.\ \pi'\\ &(\Gamma, s) =_{/Z} (\Gamma', s')\\ &\Rightarrow \Gamma \vDash s_C :: S \xrightarrow{\pi}{}^{*} t_C :: S\ \wedge\ \textit{final}\,(t)\\ &\Rightarrow \Gamma \vDash s_C' :: S \xrightarrow{\pi'}{}^{*} t_C' :: S\ \wedge\ \textit{final}\,(t')\\ &\Rightarrow \pi \downarrow_f = \pi' \downarrow_f\end{aligned}$$

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

The dependency properties defined in [13] 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:

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

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 $\mathsf{P}^{\mathcal{R}(C)}_{\mathsf{Z},f} \cap \textit{lfp}(\mathcal{R}(C)) = \emptyset$ implies $\textit{TNI}(C, \mathsf{Z}, f)$. Practically, this means that we can prove $\textit{TNI}(C, \mathsf{Z}, f)$ 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

$$\begin{aligned}\mathsf{P}^C_{\mathsf{Z},f} :=&\{\textit{InstMayDepOn}(pc, \textit{toVar}(\mathsf{z}))\ |\ \mathsf{z} \in \mathsf{Z}\\ &\quad \wedge\ C(pc) = op(\vec{x}, pc_{\textit{next}}, pre)\ \wedge\ f(op)\}\\ &\cup \{\textit{VarMayDepOn}(x_i, \textit{toVar}(\mathsf{z}))\ |\ \mathsf{z} \in \mathsf{Z} \wedge pc \in \mathsf{dom}(C)\\ &\quad \wedge\ C(pc) = (op(\vec{x}, pc_{\textit{next}}, pre))\ \wedge\ f(op)\ \wedge\ x_i \in \vec{x} \quad \}.\end{aligned}$$

The following theorem shows that $\mathsf{P}^C_{\mathsf{Z},f}$ is a security pattern for trace noninterference:

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*

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

The absence of facts from $\mathsf{P}^C_{\mathsf{Z},f}$ in $\textit{lfp}(\mathcal{R}(C))$ 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 [15].

E. 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 [27], 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 [15]. The key pillars of the soundness proof are i) that our EVM CFG semantics satisfies all conditions to be used with the slicing framework [29], 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. For Assumptions 1 and 2 we point out the existence of other sound tools [3], [24] that can check these assumptions.

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 [19] and a standard step performed by most static analysis tools [10], [27]. 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,

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

TABLE I
LARGE-SCALE EVALUATION OF HORSTIFY (**H**) AND SECURIFY (**S**).

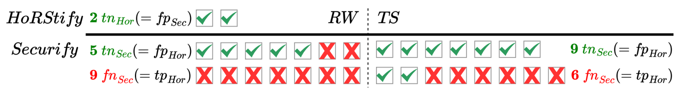


Fig. 17. Classification of mismatching results of HORSTIFY (above) and Securify (below) for the RW (left) and TS (right) property[8]. Ticks indicate correct matches (*tn*) and crosses wrong matches (*fn*) of the respective tool. tn_{Hor}/tn_{Sec}, fn_{Hor}/fn_{Sec}, tp_{Hor}/tp_{Sec}, fp_{Hor}/fp_{Sec} denote true negatives, false negatives, true positives, and true negatives of HORSTIFY/Securify, respectively.

for contracts without any preprocessing information but jump destinations needed for the CFG (cf. Section VI-A). This gives the flexibility, to enhance the precision of the analysis through the incorporation of soundly precomputed values and makes the design of sound preprocessing an orthogonal problem. There exist already works on soundly precomputing jump destinations for EVM bytecode [11], which are to be complemented with other precomputing steps in the future.

VII. 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 the prototype analyzer HORSTIFY. We do not implement the logical rules from Section VI-C directly in Soufflé (as done by Securify), but encode them in the HORST specification language [24]. 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 VI-A). 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 proceeded in two steps. First, we encoded Securify's RW violation pattern in the HORST language to execute HoRStify with this pattern and the contracts in Figures 4, 6 and 7[6]. In contrast to Securify, HORSTIFY correctly determines that these contracts do not satisfy the RW violation pattern. In addition to these corner cases, we successfully evaluated HORSTIFY on Securify's internal test suite involving 25 contracts.

Next, we conduct a large-scale evaluation of HORSTIFY and Securify on real-world contracts. To this end, we use the sanitized dataset from [24] 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 VI-D. We manually inspect all contracts on which Securify and HORSTIFY report a different result.

Table I 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 17 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 17), as we will illustrate with two examples:

Figure 18 shows a (slightly shortened) version of a contract classified as safe for TS according to Securify, but that HoRStify (correctly) reports as vulnerable. It is a lottery contract that pays out a user who manages to guess a random number (function `Guess`). The random number is generated from blockchain and transaction-specific values, including the timestamp (accessed via `now` in `RandomNumberFromSeed`). Hence, the payout in line 16 is not independent of the timestamp. Securify fails to detect this dependency due to its unsound memory abstraction (as described in Section IV-B): As Ethreum's hash function (`sha3`) reads input from the local memory, the timestamp is written to the memory where its dependencies are lost.

Figure 19 shows an example of a false positive for HORSTIFY. The contract implements a lottery where users can register (via `buyTicket`) and whenever 5 users were registered, one

[6]We did not consider the contract in Figure 3 since it concerns the must-analysis and the contract in Figure 5, which violates Assumption 2.

[8]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.

```
contract RNG {
  mapping (address => uint) nonces;
  uint public last;
  function RandomNumber() returns(uint) {
    return RandomNumberFromSeed(
      uint(sha3(block.number))^uint(sha3(now))
      ^uint(msg.sender)^uint(tx.origin)); }
  function RandomNumberFromSeed(uint seed) returns(uint) {
    nonces[msg.sender]++;
    last = seed^(uint(sha3(block.blockhash(block.number),
                            nonces[msg.sender]))
                 *0x000b0007000500030001);
    return last; }
  function Guess(uint _guess) returns (bool) {
    if (RandomNumber() == _guess) {
      if (!msg.sender.send(this.balance)) throw;
        RandomNumberGuessed(_guess, msg.sender);
        return true; }
    return false; } }
```

Fig. 18. Lottery Contract 0xaed5a41450b38fc0ea0f6f203a985653fe187d9c

```
1   contract lottery{
2     address[] public tickets;
3     function buyTicket(){
4       if (msg.value != 1/10) throw;
5       if (msg.value == 1/10)
6         tickets.push(msg.sender);
7         address(0x88a1e54971b31974b2be4d9c67546abbd0a3aa8e)
8           .send(msg.value/40);
9       if (tickets.length >= 5) runLottery(); }
10    function runLottery() internal {
11      tickets[addmod(now, 0, 5)].send((1/1000)*95);
12      runJackpot();}
13    function runJackpot() internal {
14      if(addmod(now, 0, 150) == 0)
15        tickets[addmod(now, 0, 5)].send(this.balance);
16      delete tickets; } }
```

Fig. 19. Lottery contract 0xe120100349a0b1BF826D2407E519D75C2Fe8f859

of them is selected as a winner. Despite the obvious timestamp dependency, the contract shows RW violations, which HoRStify fails to prove.[9] E.g., the `tickets` array is updated without performing a check on the sender. HoRStify does not detect this vulnerability due to its sound storage abstraction: In line 6, the caller (`msg.sender`) is appended to the `tickets` array. Since the array position to which `msg.sender` will be added cannot be statically known, HoRStify needs to assume `msg.sender` to be written to any position. When checking the size of `tickets` in line 9, the condition is considered dependent on `msg.sender` (because in the abstraction, `msg.sender` is considered to potentially affect all storage locations, including the one containing the array size). Thus, the `delete` operation in line 16 is considered dependent on `msg.sender`. One should notice, that only the unsoundness of Securify's storage abstraction, enables Securify to correctly detect the RW violation in this case.

Overall, based on our evaluation results, we can bound the precision loss of HoRStify w.r.t. Securify. More concretely, when considering that Securify has a specificity[10] 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 17, 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.

We refer to `horstify.org` for more information about HoRStify.

VIII. Related Work

Existing approaches to enforce the correctness of Ethereum smart contracts can be broadly categorized into analyses at design time and analyses at runtime. The latter include methods like runtime monitoring [8], [28] or information flow control mechanisms [5]. Such dynamic analysis approaches, however, have limited applicability to the Ethereum blockchain, since they either require fundamental updates to the workings of the EVM or impose tremendous costs in terms of gas. Static analyses, in contrast, verify smart contracts at design time before they become immutable objects on the blockchain. Most static analyzers are bug-finding tools (such as Oyente [22], EthBMC [9], and Maian [23]) 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 dual objective of bug-finding is to prove a smart contract secure. Analyzers following this objective do not only aim at producing a low number of false negatives in practice but to give provable guarantees for their analysis result, e.g., that a contract flagged as safe is guaranteed to enjoy a corresponding security property. The only example of a tool, which comes with a provable soundness claim, so far, is the analyzer eThor [24], whose analysis relies on abstract interpretation.

Symbolic execution and abstract interpretation have in common to target properties that can be decided for a finite prefix of a single (yet arbitrary) execution trace of a smart contract (so-called *reachability properties*). However, many generic security properties for smart contracts (as defined in [13]) require comparing *two* execution traces from different initial configurations and 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. But finding such a meaningful overapproximation, which does not result in an intolerable precision loss, is not always possible. In [13], 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. However, trace noninterference properties still concern two execution traces and hence cannot be verified using eThor. HoRStify (inspired by the unsound Securify tool [27]) devises a different analysis technique, which immediately accommodates the analysis of trace noninterference. As opposed to the analysis

[9]Note that this is not a soundness issue since the soundness of HoRStify ensures that independencies can be proven. In the case of violation patterns as RW the independence constitutes an unwanted effect and hence, we can only use it to prove the vulnerability of a contract, not its safety.

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

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 (within the scope of HoRSTIFY) and a reachability property (within the scope of eThor). Other generic security properties from [13] 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 [6], [7] cannot be checked by either HoRSTIFY or eThor.

IX. 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 [27], we uncover conceptual soundness issues of the tool, so we replace the underlying analysis and spell out formal soundness guarantees. Even though we need to tighten the scope of the Securify analysis (removing the must-analysis) for achieving 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.

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] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C Myers. Compositional security for reentrant applications. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1249–1267. IEEE, 2021.

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

[7] 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.

[8] Joshua Ellul and Gordon J Pace. Runtime verification of ethereum smart contracts. In *2018 14th European Dependable Computing Conference (EDCC)*, pages 158–163. IEEE, 2018.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] Haijun Wang, Yi Li, Shang-Wei Lin, Lei Ma, and Yang Liu. Vultron: catching vulnerable smart contracts once and for all. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 1–4. IEEE, 2019.

[29] 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.

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

[31] 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.

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