Securify: Practical Security Analysis of Smart Contracts

Petar Tsankov
ETH Zurich
petar.tsankov@inf.ethz.ch

Andrei Dan
ETH Zurich
andrei.dan@inf.ethz.ch

Dana Drachsler-Cohen
ETH Zurich
dana.drachsler@inf.ethz.ch

Arthur Gervais*
Imperial College London
a.gervais@imperial.ac.uk

Florian Bünzli
ETH Zurich
fbuenzli@student.ethz.ch

Martin Vechev
ETH Zurich
martin.vechev@inf.ethz.ch

ABSTRACT

Permissionless blockchains allow the execution of arbitrary programs (called *smart contracts*), enabling mutually untrusted entities to interact without relying on trusted third parties. Despite their potential, repeated security concerns have shaken the trust in handling billions of USD by smart contracts.

To address this problem, we present Securify, a security analyzer for Ethereum smart contracts that is scalable, fully automated, and able to prove contract behaviors as safe/unsafe with respect to a given property. Securify's analysis consists of two steps. First, it symbolically analyzes the contract's dependency graph to extract precise semantic information from the code. Then, it checks compliance and violation patterns that capture sufficient conditions for proving if a property holds or not. To enable extensibility, all patterns are specified in a designated domain-specific language.

Securify is publicly released, it has analyzed $> 18K$ contracts submitted by its users, and is regularly used to conduct security audits by experts. We present an extensive evaluation of Securify over real-world Ethereum smart contracts and demonstrate that it can effectively prove the correctness of smart contracts and discover critical violations.

KEYWORDS

Smart contracts; Security analysis; Stratified Datalog

1 INTRODUCTION

Blockchain platforms, such as Nakamoto's Bitcoin [43], enable the trade of crypto-currencies between mutually mistrusting parties. To eliminate the need for trust, Nakomoto designed a peer-to-peer network that enables its peers to agree on the trading transactions. Buterin [24] identified the applicability of decentralized computation beyond trading, and designed the Ethereum blockchain which supports the execution of programs, called smart contracts, written in Turing-complete languages. Smart contracts have shown to be applicable in many domains including financial industry [8], public sector [11] and cross-industry [9].

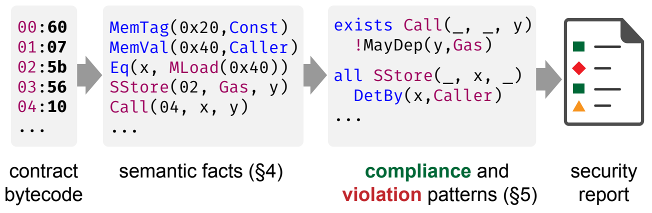
The increased adoption of smart contracts demands strong security guarantees. Unfortunately, it is challenging to create smart contracts that are free of security bugs. As a consequence, critical vulnerabilities in smart contracts are discovered and exploited every few months [2, 3, 6, 7, 10, 26]. In turn, these exploits have led to losses reaching millions worth of USD in the past few years: 150M were stolen from the popular DAO contract in June 2016 [6], 30M were stolen from the widely-used Parity multi-signature wallet in July 2017 [10], and few months later 280M were frozen due to a bug in the very same wallet [13]. It is apparent that effective security checkers for smart contracts are urgently needed.

Key Challenges. The main challenge in creating an effective security analyzer for smart contracts is the Turing-completeness of the programming language, which renders automated verification of arbitrary properties undecidable. To address this issue, current automated solutions tend to rely on fairly generic testing and symbolic execution methods (e.g., Oyente [39] and Mythril [16]). While useful in some settings, these approaches come with several drawbacks: *(i)* they can miss critical violations (due to under-approximation), *(ii)* yet, can also produce false positives (due to imprecise modeling of domain-specific elements [30]), and *(iii)* they can fail to achieve sufficient code coverage on realistic contracts (Oyente achieves only 20.2% coverage on the popular Parity wallet [17]). Overall, these drawbacks place a significant burden on their users, who must inspect all reports for false alarms and worry about unreported vulnerabilities. Indeed, many security properties for smart contracts are inherently difficult to reason about directly. A viable path to addressing these challenges is building an automated verifier that targets important domain-specific properties [15]. For example, recent work [31] focuses solely on identifying reentrancy issues in smart contracts [5].

Domain-Specific Insight. A key observation of this work is that it is often possible to devise precise patterns expressed on the contract's data-flow graph in a way where a match of the pattern implies either a violation or satisfaction of the original security property. For example, 90.9% of all calls in Ethereum smart contracts can be proved free of the infamous DAO bug [6] by matching a pattern stating that calls are not followed by writes to storage. The reason why it is possible to establish such a correspondence is that violations of the original property in real-world contracts tend to often violate a much simpler property (captured by the pattern). Indeed, in terms of verification, a key benefit in working with patterns, instead of with their corresponding property, is that patterns are substantially more amenable to automated reasoning.

Securify: Domain-specific Verifier. Based on the above insight, we developed Securify, a lightweight and scalable security verifier for Ethereum smart contracts. The key technical idea is to define two kinds of patterns that mirror a given security property: *(i)* compliance patterns, which imply the satisfaction of the property, and *(ii)* violation patterns, which imply its negation. To check these patterns, Securify symbolically encodes the dependence graph of the contract in stratified Datalog [50] and leverages off-the-shelf

*Work done while at ETH Zurich



```
00:60
01:07
02:5b
03:56
04:10
...
```

```
MemTag(0x20,Const)
MemVal(0x40,Caller)
Eq(x, MLoad(0x40))
SStore(02, Gas, y)
Call(04, x, y)
...
```

```
exists Call(_, _, y)
  !MayDep(y,Gas)
all SStore(_, x, _)
  DetBy(x,Caller)
...
```

Figure 1: Securify's approach is based on automatic inference of semantic program facts followed by checking of compliance and violation security patterns over these facts.

scalable Datalog solvers to efficiently (typically within seconds) analyze the code. To ensure extensibility, all patterns are expressed in a designated domain-specific language (DSL).

In Fig. 1, we illustrate the analysis flow of Securify. Starting with the contract's bytecode (or source code, which can be compiled to bytecode), Securify derives semantic facts inferred by analyzing the contract's dependency graph and uses these facts to check a set of compliance and violation patterns. Based on the outcome of these checks, Securify classifies all contract behaviors into violations (◆), warnings (▲), and compliant (■), as abstractly illustrated in Fig. 2. Here, the large box depicts all contract behaviors, partitioned into safe (which satisfy the property) and unsafe ones (which violate it). Securify reports as violations (◆) all behaviors matching the violation pattern, and as warnings (▲) all remaining behaviors not matched by the compliance pattern.

Reduced Manual Effort. Compared to existing symbolic analyzers for smart contracts, Securify reduces the required effort to inspect reports in two ways. First, existing analyzers do not report definite violations (they conflate ◆ and ▲), and thus require users to manually classify *all* reported vulnerabilities into true positives (found in the red box) or false positives (found in the green box). In contrast, Securify automatically classifies behaviors guaranteed to be violations (marked with ◆). Hence, the user only needs to manually classify the warnings (▲) as true or false positives.

As we show in our evaluation, the approach of using both violation and compliance patterns reduces the warnings a user needs to inspect manually by 65.9%, and even up to 99.4% for some properties. Second, existing analyzers fail to report unsafe behaviors (sometimes up to 72.9%), meaning users may have to manually inspect portions of the code that are not covered by the analyzer. In contrast, Securify reports all unsafe behaviors.

Auditing Smart Contracts. Securify is publicly available at https://securify.ch and has analyzed $> 18K$ contracts submitted by its users. Over the last year, we have also extensively used Securify to perform 38 detailed commercial audits of smart contracts (other auditors have also used Securify), iteratively improving the approach and adding more patterns. Indeed, the design and implementation of Securify have greatly benefited from this experience.

In terms of the actual audit process, our approach (and we believe that of other auditors) has been to run all available tools and then to manually inspect the reported vulnerabilities so to assess their severity. For instance, while Securify covers a number of important properties (the full version supports 18 properties), symbolic

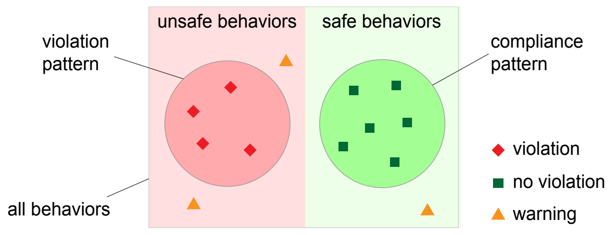


Figure 2: Securify uses compliance and violation patterns to guarantee that certain behaviors are safe and, respectively, unsafe. The remaining behaviors are reported as warnings (to avoid missing errors).

execution tools have better support for numerical properties (e.g., overflow). Our finding was that Securify was particularly helpful in auditing larger contracts, which are challenging to inspect with existing solutions for the reasons listed earlier. Overall, we believe Securify is a pragmatic and valuable point in the space of analyzing smart contracts due to its careful balance of scalability, guarantees, and precision.

Main Contributions. To summarize, our main contributions are:

- A decompiler that symbolically encodes the dependency graph of Ethereum contracts in Datalog (Section 4).
- A set of compliance and violation security patterns that capture sufficient conditions to prove and disprove practical security properties (Section 5).
- An end-to-end implementation, called Securify, which fully automates the analysis of contracts (Section 6).
- An extensive evaluation over existing Ethereum smart contracts showing that Securify can effectively prove the correctness of contracts and discover violations (Section 7).

2 MOTIVATING EXAMPLES

In this section, we motivate the problem we address through two real-world security issues that affected $\approx$ 200 millions worth of USD in 2017. We describe the underlying security properties and the challenges involved in proving whether a contract satisfies/violates them. We also describe how Securify discovers both vulnerabilities with appropriate violation patterns.

2.1 Stealing Ether

In Fig. 3, we show an implementation of a wallet. The code is written in Solidity [18], a popular high-level language for writing Ethereum smart contracts. We remark that this wallet is a simplified version of Parity's multi-signature wallet, which allowed an attacker to steal 30 million worth of USD in July 2017.

The wallet has a field `owner`, which stores the address of the wallet's owner. Further, the contract has a function `initWallet`, which takes as argument an address `_owner` and initializes the field `owner` with it. This function is called by the constructor (not shown in Fig. 3), and was assumed not to be accessible otherwise [10]. Finally, the contract has a function `withdraw`, which takes as argument an unsigned integer `_amount`. The function checks if the transaction sender's address (returned by `msg.sender`) equals that

```
contract OwnableWallet {
  address owner;

  // called by the constructor
  function initWallet(address _owner) {
    owner = _owner; // any user can change owner
    // more setup
  }

  // function that allows the owner to withdraw ether
  function withdraw(uint _amount) {
    if (msg.sender == owner) {
      owner.transfer(_amount);
    }
  }
  // ...
}
```

Figure 3: A vulnerable wallet that allows any user to withdraw all ether stored in it.

```
contract Wallet {
  // fixed address of the wallet library
  address constant walletLibrary = ...;

  // function that receives ether
  function deposit() payable {
    log(msg.sender, msg.value);
  }

  // function for withdrawing ether
  function withdraw() {
    walletLibrary.delegatecall(msg.data);
  }

  // ...
} // No guaranteed ether transfer
```

Figure 4: A wallet that delegates functionality to a library contract `walletLibrary`.

of the contract's owner (stored in the field `owner`). If this check succeeds, it transfers `_amount` ether to the owner with the statement `owner.transfer(_amount)`; otherwise, no ether is transferred. The `withdraw` function ensures that only the owner can withdraw ether from the wallet.

Attack. The wallet shown in Fig. 3 has a critical security flaw: any user could actually call the `initWallet` function and store an arbitrary address in the field `owner`. An attacker can, therefore, steal all ether stored in the wallet in two steps. First, the attacker calls the function `initWallet`, passing her own address as argument. Second, the attacker calls the function `withdraw`, passing as argument the amount of ether stored in the wallet. We remark that in the attack on Parity's wallet, to perform the first step the attacker exploits a fallback mechanism to call the `initWallet` function; we omit these details for simplicity and refer the reader to [10] for details on the actual attack.

Security Property. The underlying security problem that allowed the attacker to steal ether is that the security-critical field `owner` is universally writable by *any* Ethereum user. This security issue mirrors a more general property stipulating that the write to the `owner` field is restricted, in the sense that not all users can make a transaction that writes to this field. To show that this property is satisfied, we need to demonstrate that some user cannot send a transaction that modifies the `owner` field. Conversely, to show a violation, we need to prove that all users can send a transaction that modifies the `owner` field. Proving both satisfaction and demonstrating violations of this property is nontrivial due to the enormous space of possible users and transactions that they can make.

Detection. To discover this security issue, SECURIFY provides a violation pattern that is matched if the execution of the assignment `owner = _owner`, highlighted in red in Fig. 3, does not depend on the value returned by the caller instruction (which returns the address of the transaction sender). To check this pattern, SECURIFY infers data- and control-flow dependencies by analyzing the contract's dependency graph; cf. [35]. Here, SECURIFY infers that the assignment `owner = _owner` does not depend on the caller instruction, which implies that the assignment is reachable by any user. In Section 3, we provide more details on this violation pattern and further details on how SECURIFY uses it to detect the vulnerability.

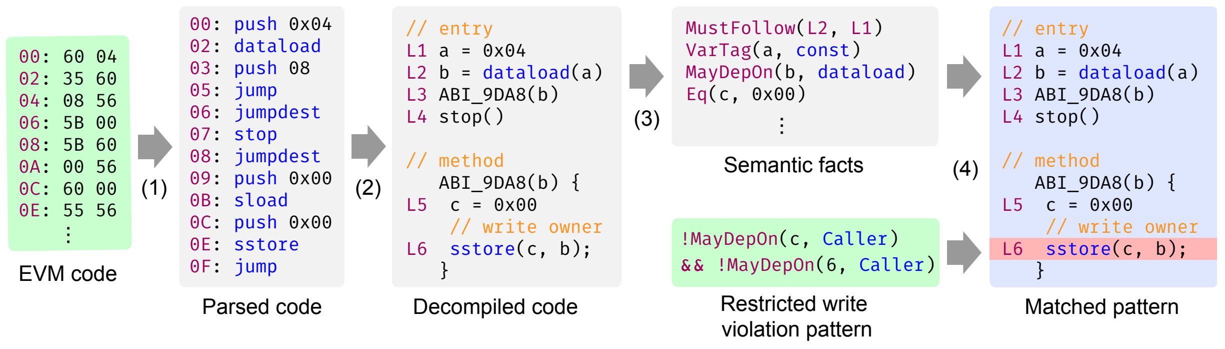
We remark that some symbolic checkers perform imprecise checks of similar properties, which result in both false positives and false negatives. For instance, as we show in Fig. 13 of our evaluation later, Mythril [16] has about 65% false negatives when checking a similar property stipulating that not all users may trigger a particular ether transfer.

2.2 Frozen Funds

In Fig. 4, we show a wallet implementation which suffers from a security issue that froze millions worth of USD in November 2017. This wallet has a field, `walletLibrary`, which stores the address of a contract implementing common wallet functionality. Further, it has a function `deposit`, marked as *payable*, which means users can send ether to the contract by calling this function. The function `deposit` logs the amount of ether (identified by `msg.value`) sent by the transaction sender (identified by `msg.sender`). Finally, the contract has a function `withdraw`, which delegates all calls to the wallet library. That is, the statement `walletLibrary.delegatecall(msg.data)` results in executing the `withdraw` method of the wallet library in the context of the current wallet.

Attack. Ethereum contracts can be removed from the blockchain using a designated `kill` instruction. If an attacker can remove the wallet library from the blockchain, then the funds in the wallet cannot be extracted from the wallet. This is because the wallet relies on the library smart contract to withdraw ether. In November 2017, a popular wallet library was removed from the blockchain, effectively freezing $\approx$ 280 million worth of USD [7].

Security Property. The underlying security problem with this wallet is that it allows users to deposit ether, but it cannot guarantee that the ether can be transferred out of the contract, since the transfer depends on a library. To discover that the wallet has this problem, we must prove two facts: *(i)* users can deposit ether and



```
00: 60 04
02: 35 60
04: 08 56
06: 5B 00
08: 5B 60
0A: 00 56
0C: 60 00
0E: 55 56
  ⋮
```

```
00: push 0x04
02: dataload
03: push 08
05: jump
06: jumpdest
07: stop
08: jumpdest
09: push 0x00
0B: sload
0C: push 0x00
0E: sstore
0F: jump
```

```
// entry
L1 a = 0x04
L2 b = dataload(a)
L3 ABI_9DA8(b)
L4 stop()

// method
   ABI_9DA8(b) {
L5  c = 0x00
    // write owner
L6  sstore(c, b);
   }
```

```
MustFollow(L2, L1)
VarTag(a, const)
MayDepOn(b, dataload)
Eq(c, 0x00)
     ⋮
```

```
!MayDepOn(c, Caller)
&& !MayDepOn(6, Caller)
```

```
// entry
L1 a = 0x04
L2 b = dataload(a)
L3 ABI_9DA8(b)
L4 stop()

// method
   ABI_9DA8(b) {
L5  c = 0x00
    // write owner
L6  sstore(c, b);
   }
```

Figure 5: High-level flow illustrating how Securify finds the unrestricted write to the `owner` field of the contract from Fig. 3. The input (EVM bytecode and security patterns) is highlighted in green, the output (in our example, a violated instruction) is highlighted in red, and gray boxes represent intermediate analysis artifacts. Securify proceeds in three steps: (1) it decompiles the contract's EVM bytecode into a static-single assignment form, (2) it infers semantics facts about the contract, and (3) it matches the violation pattern of the restricted write property on the `sstore` instruction that writes to the `owner` field.

(ii) the contract has no ether transfer instructions (i.e., call) with non-zero amount of ether. Note that if the contract only transfers out ether through libraries, the second requirement is met.

Detection. To discover this vulnerability, Securify's violation pattern checks the conjunction of two facts. First, to prove that users can deposit ether, Securify checks whether there is a stop instruction whose execution does not depend on the ether transferred being zero. Assuming that the stop instruction is reachable for some transaction, this implies that a user can reach it *with a positive ether amount*, resulting in a deposit of ether to the contract. Second, Securify checks whether for all call instructions, the amount of ether extracted from the contract is zero. The conjunction of these two facts implies that ether can be locked in the contract.

3 THE SECURIFY SYSTEM

In the previous section, we illustrated that while security issues in smart contracts are complex, they can be often captured with semantic facts inferred from the code. In this section, we describe the Securify system, which builds on this idea to prove and disprove security properties of smart contracts. We accompany this section with the example of how Securify detects the unrestricted write to the `owner` field in the wallet contract (Fig. 3). Fig. 5 summarizes the main steps.

Inputs to Securify. The input to Securify is the EVM bytecode of a contract and a set of security patterns, specified in our designated domain-specific language (DSL). Securify can also take as input contracts written in Solidity (not shown in Fig. 5), which are compiled to EVM bytecode before proceeding with the analysis. There are two kinds of security patterns: compliance and violation patterns, which capture sufficient conditions to ensure that a contract satisfies and, respectively, violates a given security property.

Fig. 5 illustrates the input to Securify in green boxes, which show part of the EVM bytecode of the wallet contract (only the part necessary to illustrate the vulnerability) and the violation pattern of the *restricted write* property. Intuitively, the pattern is matched if there is a write that is *not* restricted.

To discover the unrestricted write in the contract, Securify proceeds with the following three steps.

Step 1: Decompiling EVM Bytecode. Securify first transforms the EVM bytecode provided as input into a stackless representation in static-single assignment form (SSA). For example, in Fig. 5, for the stack expression `push 0x04`, Securify introduces a local variable `a` and an assignment statement `a = 4`. In addition to removing the stack, Securify identifies methods. For example, the method `ABI_9DA8`, shown in Fig. 5, corresponds to the `initOwner` method of the wallet contract, shown in Fig. 3. After decompilation, Securify performs partial evaluation to resolve memory and storage offsets, jump destinations, all of which are important for precisely analyzing the code statically. We describe these optimizations in Section 6.

Step 2: Inferring Semantic Facts. After decompilation, Securify analyzes the contract to infer semantic facts, including data- and control-flow dependencies, which hold over all behaviors of the contract. For example, the fact *MayDepOn*(`b`, dataload), shown in Fig. 5, captures that the value of variable `b` may depend on the value returned by the instruction dataload. Further, the fact *Eq*(`c`, 0) captures that variable `c` equals the constant 0.

Securify's derivation of semantic facts is specified declaratively in stratified Datalog and is fully automated using existing scalable engines [36]. Key benefits of the declarative approach are: *(i)* inference rules concisely capture abstract reasoning about different components (e.g., contract storage), *(ii)* more facts and inference rules can be easily added, and *(iii)* inference rules are specified in a modular way (e.g., memory analysis is specified independently of contract storage analysis). We list the semantic facts that Securify derives, along with the inference rules, in Section 4.

Step 3: Checking Security Patterns. After obtaining the semantics facts, Securify checks the set of compliance and violation security patterns, given as input. These patterns are written in a specialized domain-specific language (DSL), which enables security experts to extend our built-in set of patterns with their customized patterns. Our DSL is a fragment of logical formulas over the semantic facts inferred by Securify. To detect the vulnerability in the contract of Fig. 3, Securify matches the violation pattern (given as input) on the sstore(c, b) instruction at label l6 in Fig. 5. In sstore(c, b), c is the storage offset of the `owner` field, and b is the value to store. The violation pattern matches if there exists some sstore instruction for which both the storage offset, denoted X, and the execution of this instruction, identified by its label L, do not depend on the result of the caller instruction in *any possible execution* of the contract. Since the instruction caller retrieves the address of the transaction sender, matching this pattern implies that any user can reach this sstore and change the value of `owner`. In our DSL, where negation is encoded by ¬ and conjunction by ∧, this pattern is encoded as:

$$\textit{some}\ \mathrm{sstore}(L, X, \_).\ \neg \textit{MayDepOn}(X, \mathrm{caller}) \wedge \neg \textit{MayDepOn}(L, \mathrm{caller})$$

Securify's DSL is important for extensibility: adding new security patterns amounts to specifying them in this DSL. To illustrate the expressiveness of the DSL, in Section 5, we present a range of security patterns for important properties, such as restricted writes, exception handling, ether liquidity, input validation, and others.

We remark that contract-specific patterns are sometimes added by security experts while conducting security audits. For example, it is often required to check for the absence of undesirable dependencies, such as: only the owner can modify certain values in the storage, or to ensure that the result of a specific arithmetic expression does not depend on the division instruction (which may cause undesirable integer rounding effects). We illustrate how such contract-specific patterns are specified in the DSL in Section 5.

Output of Securify. For any match of a violation pattern, Securify outputs the instruction that caused the pattern to match. In our example, it highlights the instruction sstore(c, b). We remark that the offset of this instruction can be easily mapped to its corresponding line in the Solidity code, if the source code is provided. Further, for any property for which neither the violation nor the compliance pattern is matched, Securify outputs a warning, indicating that it failed to prove or disprove the property.

Limitations. We briefly summarize several limitations of Securify. First, the current version of Securify cannot reason about numerical properties, such as overflows. To address this limitation, we plan to extend Securify with numerical analysis (e.g., using ELINA [48]), which would not only improve the precision of Securify but also enable the checking of numerical properties.

Second, Securify does not reason about reachability, and assumes that all instructions in the contract are reachable. This assumption is necessary to establish a formal correspondence between the security properties supported by Securify and the patterns used to prove and disprove them. For instance, in our example, Securify assumes that the matched sstore instruction is reachable by some execution (otherwise, there is no violation).

Finally, the properties we consider capture violations that can often, but not always, be exploited by attackers. For example, there are fields in the contract that must be universally writable by all users. To address this, security experts can write contract-specific patterns in Securify's DSL (e.g., to specify which fields are sensitive).

4 SEMANTIC FACTS

In this section, we present the automated inference of control- and data-flow dependencies that Securify employs. The facts inferred in this process are called *semantic facts* and are later used for checking security properties. We begin with the background necessary for understanding this analysis: the EVM instruction set and stratified Datalog. We then introduce the semantic facts derived by Securify and the declarative inference rules, specified in stratified Datalog, used to derive them.

4.1 Background

In this section, we provide the necessary background.

4.1.1 Ethereum Virtual Machine (EVM). Smart contracts are executed on a *blockchain*. A contract executes when a user submits a *transaction* that specifies the contract, the method to run, and the method's arguments (if any). When the transaction is processed, it is added to a new block, which is appended to the blockchain. Contracts can access a volatile memory and non-volatile storage. The EVM instruction set (over which contracts are written) supports a few dozen opcodes. Securify handles all EVM opcodes; we present the most relevant ones below. Note that many of the opcodes (such as push, dup, etc.) are eliminated when Securify decompiles the EVM bytecode to its stackless representation. The relevant instructions are:

- Arithmetic operations and comparisons: e.g., add, mul, lt, eq. In the rest of the paper, we write op to denote any of these operations.
- Cryptographic hash functions: e.g., sha3.
- Environmental information: e.g., balance returns the balance of a contract, caller is the identity of the transaction sender, callvalue is the amount of ether specified to be transferred by the transaction.
- Block information: e.g., number, timestamp, gaslimit.
- Memory and storage operations: mload, mstore, sstore, sload load/store data from the memory/contract storage.
- System operations: e.g., call, which transfers ether, and takes two arguments: receiver address and amount of ether to transfer (in fact, call takes seven arguments; we consider here only those that are relevant for the rest of the paper).
- Control-flow instructions: e.g., goto, which encodes conditional jumps across instructions.

For the complete set of instructions, along with their formal description, we refer the reader to [52].

4.1.2 Stratified Datalog. Stratified Datalog is a declarative logic language, which enables to write *facts* (predicates) and *rules* to infer facts. We next briefly overview its syntax and semantics.

Syntax. We present Datalog's syntax in Fig. 6. A Datalog program consists of one or more rules, denoted $\overline{r}$. A rule r consists of a head a, and a body, $\overline{l}$, consisting of literals, separated by commas. The head, also called an atom, is a predicate over zero or more terms, denoted $\overline{t}$, comma-separated. A literal l is a predicate or its negation. As a convention, we write Datalog variables in upper case and constants in lower case. A Datalog program is *well-formed* if

(Program)	P	::=	$\bar{r}$	*(Predicates)*	p, q	$\in$	$\mathcal{P}$
(Rule)	r	::=	$a \Leftarrow \bar{l}$	*(Term)*	t	$\in$	$\mathcal{V} \cup C$
(Atom)	a	::=	$p(\bar{t})$	*(Datalog variables)*	X, Y	$\in$	$\mathcal{V}$
(Literal)	l	::=	$a \mid \neg a$	*(Constants)*	x, y	$\in$	C

Figure 6: Syntax of stratified Datalog.

for any rule $a \Leftarrow \bar{l}$, we have $vars(a) \subseteq vars(\bar{l})$, where $vars(\bar{l})$ returns the set of variables in $\bar{l}$.

A Datalog program P is *stratified* if its rules can be partitioned into strata $P_1, \ldots, P_n$ such that if a predicate p occurs in a positive (negative) literal in the body of a rule in P_i, then all rules with p in their heads are in a stratum P_j with $j \leq i$ ($j < i$). Stratification ensures that predicates that appear in negative literals are fully defined in lower strata.

Semantics. Let $\mathcal{A} = \{p(\bar{t}) \mid \bar{t} \subseteq C\}$ (where $\bar{t}$ is a list of terms separated by commas) denote the set of all ground (i.e., variable-free) atoms; we refer to these as *facts*. An interpretation $A \subseteq \mathcal{A}$ is a set of facts. The complete lattice $(\mathcal{P}(\mathcal{A}), \subseteq, \cap, \cup, \emptyset, \mathcal{A})$ partially orders the set of interpretations $\mathcal{P}(\mathcal{A})$.

Given a substitution $\sigma \in \mathcal{V} \to C$, mapping variables to constants, and an atom a, we write $\sigma(a)$ for the fact obtained by replacing the variables in a according to σ. For example, $\sigma(p(X))$ returns the fact $p(\sigma(X))$. Given a program P, its consequence operator $T_P \in \mathcal{P}(\mathcal{A}) \to \mathcal{P}(\mathcal{A})$ is defined as:

$$T_P(A) = \{\sigma(a) \mid (a \Leftarrow l_1 \ldots l_n) \in P, \forall l_i \in \bar{l}.\ A \vdash \sigma(l_i)\}$$

where $A \vdash \sigma(a)$ if $\sigma(a) \in A$ and $A \vdash \sigma(\neg a)$ if $\sigma(a) \notin A$.

An input for P is a set of facts constructed using P's extensional predicates, i.e., those that appear only in the rule bodies. Let P be a program with strata $P_1, \ldots, P_n$ and I be an input for P. The model of P for I, denoted by $[\![P]\!]_I$, is M_n, where $M_0 = I$ and $M_i = \bigcap\{A \in \text{fp } T_{P_i} \mid M_{i-1} \subseteq A\}$ is the smallest fixed point of T_{P_i} that is greater than or equal to the lower stratum's model M_{i-1}.

4.2 Facts and Inference Rules

Securify first extracts a set of base facts that hold for every instruction. These base facts constitute a Datalog input that is fed to a Datalog program to infer additional facts about the contract. We use the term *semantic facts* to refer to the facts derived by the Datalog program. All program elements that appear in the contract, including instruction labels, variables, fields, string and integer constants, are represented as constants in the Datalog program.

Base Facts. The base facts of our inference engine describe the instructions in the contract's control-flow graph (CFG). The base facts take the form of instr$(L, Y, X_1, \ldots, X_n)$, where instr is the instruction name, L is the instruction's label, Y is the variable storing the instruction result (if any), and $X_1, \ldots, X_n$ are variables given to the instruction as arguments (if any). For example, the instruction l1: `a = 4` (from Fig. 5) is encoded to assign(l1, a, 4). Further, the instruction l6: sstore(c, b), where the variable `c` is known to be equal to the constant `0` at compile time, is encoded to sstore(l6, 0, b); if the value of the variable `c` could not be determined at compile time, then the instruction would be encoded to sstore(l6, $\top$, b), where $\top$ is a Datalog constant that encodes that the value of `c` is unknown.

Semantic fact	Intuitive meaning
Flow Dependencies	
MayFollow(L_1, L_2)	Instruction at label L_2 may follow that at label L_1.
MustFollow(L_1, L_2)	Instruction at label L_2 must follow that at label L_1.
Data Dependencies	
MayDepOn(Y, T)	The value of Y may depend on tag T.
Eq(Y, T)	The values of Y and T are equal.
DetBy(Y, T)	For different values of T the value of Y is different.

Figure 7: The semantic facts: L_1 and L_2 are labels, Y is a variable, and T is a tag (a variable or a label).

The base facts of consecutive instructions are expressed by a predicate defined over labels called *Follow*. For every two labels, L_1 and L_2, whose instructions are consecutive in the CFG (either in the same basic block or in linked basic blocks), we have the base fact *Follow*(L_1, L_2). An example, *Follow* fact derived for the contract, shown in Fig 5, is *Follow*(l1, l2). The join of then/else branches is captured by a predicate *Join*(L_1, L_2), which encodes that the two branches that originate at an instruction goto(L_1, X, L_3), located at label L_1, are joined (i.e., they are merged into a single path) at label L_2. Using the base facts described above, Securify computes two kinds of semantic facts: *(i)* flow-dependency predicates, which capture instruction dependencies according to the contract's CFG, and *(ii)* data-dependency predicates; see Fig. 7.

Flow-Dependency Predicates. The flow predicates we consider are *MayFollow* and *MustFollow*, both are defined over pairs of labels and are inferred from the contract's CFG. The intuitive meaning (also summarized in Fig. 7) is:

- *MayFollow*(L_1, L_2) holds for L_1 and L_2 if both are in the same basic block and L_2 follows L_1, or there is a path from the basic block of L_1 to the basic block of L_2.
- *MustFollow*(L_1, L_2) holds if both are in the same basic block and L_2 follows L_1, or any path to the basic block of L_2 passes through the basic block of L_1.

To infer the *MayFollow* and *MustFollow* predicates, we use the *Follow*(L_1, L_2) input fact which holds if L_2 immediately follows $L1$ in the CFG. Namely, the predicate *MayFollow* is defined with the following two Datalog rules:

$$\begin{array}{lcl} \mathit{MayFollow}(L_1, L_2) & \Leftarrow & \mathit{Follow}(L_1, L_2) \\ \mathit{MayFollow}(L_1, L_2) & \Leftarrow & \mathit{MayFollow}(L_1, L_3),\ \mathit{Follow}(L_3, L_2) \end{array}$$

The first rule is interpreted as: if *Follow*(L_1, L_2) holds (i.e., it is contained in the Datalog input), then the predicate *MayFollow*(L_1, L_2) is derived (i.e., it is added to the fixed-point). The second rule is interpreted as: if both *MayFollow*(L_1, L_3) and *Follow*(L_3, L_2) hold, then *MayFollow*(L_1, L_2) is derived. Note that if *MayFollow*(L_1, L_2) is not derived in the fixed-point (at the end of the fixed-point computation), then the instruction at label L_2 *does not* appear after the instruction at label L_1, *in any execution* of the contract.

The inference rules for *MustFollow* are defined similarly, with a special attention to the join points in the CFG.

Data-dependency may-analysis

$$\begin{aligned}
&MayDepOn(Y, X) &&\Leftarrow \mathrm{assign}(\_, Y, X)\\
&MayDepOn(Y, T) &&\Leftarrow \mathrm{assign}(\_, Y, X), MayDepOn(X, T)\\
&MayDepOn(Y, T) &&\Leftarrow \mathrm{op}(\_, Y, \ldots, X, \ldots), MayDepOn(X, T)\\
&MayDepOn(Y, T) &&\Leftarrow \mathrm{mload}(L, Y, O), isConst(O), MemTag(L, O, T)\\
&MayDepOn(Y, T) &&\Leftarrow \mathrm{mload}(L, Y, O), \neg isConst(O), MemTag(L, \_, T)\\
&MayDepOn(Y, T) &&\Leftarrow \mathrm{mload}(L, Y, O), MemTag(L, \top, T)\\
&MayDepOn(Y, T) &&\Leftarrow \mathrm{assign}(L, Y, \_), Taint(\_, L, X), MayDepOn(X, T)
\end{aligned}$$

Memory analysis inference

$$\begin{aligned}
&MemTag(L, O, T) &&\Leftarrow \mathrm{mstore}(L, O, X), isConst(O), MayDepOn(X, T)\\
&MemTag(L, \top, T) &&\Leftarrow \mathrm{mstore}(L, O, X), \neg isConst(O), MayDepOn(X, T)\\
&MemTag(L, O, T) &&\Leftarrow Follow(L_1, L), MemTag(L_1, O, T),\\
& && \quad \neg ReassignMem(L, O)\\
&ReassignMem(L, O) &&\Leftarrow \mathrm{mstore}(L, O, \_), isConst(O)
\end{aligned}$$

Implicit control-flow analysis

$$\begin{aligned}
&Taint(L_1, L_2, X) &&\Leftarrow \mathrm{goto}(L_1, X, L_2)\\
&Taint(L_1, L_2, X) &&\Leftarrow \mathrm{goto}(L_1, X, \_), Follow(L_1, L_2)\\
&Taint(L_1, L_2, X) &&\Leftarrow Taint(L_1, L_3, X), Follow(L_3, L_2), \neg Join(L_1, L_2)
\end{aligned}$$

Figure 8: Partial inference rules for *MayDepOn*: the Datalog variable X ranges over contract variables, L ranges over instruction labels, $\top$ represents an unknown offset, and T ranges over tags.

Data-Dependency Predicates. The dependency predicates we consider are *MayDepOn*, *Eq*, and *DetBy*. The intuitive meaning of them (also summarized in Fig. 7) is:

– *MayDepOn*(Y, T) is derived if the value of variable Y depends on the *tag* T. Here, the variable T ranges over tags, which can be a contract variable (e.g., x) or an instruction (e.g., timestamp). For example, *MayDepOn*(Y, X) means that the value of variable Y may change if the value of X changes, while *MayDepOn*$(Y,$ timestamp$)$ means that Y may change if the instruction timestamp returns a different value.

– *Eq*(Y, T) indicates that the values of Y and T are identical. For example, given fact assign$(l, x,$ caller$)$, which stores the sender's address at variable x, we have *Eq*$(x,$ caller$)$.

– *DetBy*(Y, T) indicates that a different value of T guarantees that the value of Y changes. For example, *DetBy*$(x,$ caller$)$ is derived if we have the fact sha3$(l,$ `x`, `start`, `len`$)$, which returns the hash of the memory segment starting at offset `start` and ending at offset `start` + `len`, if any part of this memory segment is determined by caller. Note that *Eq*(Y, T) implies that *DetBy*(Y, T) also holds.

The Datalog rules defining these data-dependency predicates are given in Fig. 8. To avoid clutter in the rules, we use the wildcard symbol (_) in place of variables that appear only once in the rule; for example, we write *MayDepOn*$(Y, X) \Leftarrow$ assign$(_, Y, X)$ instead of *MayDepOn*$(Y, X) \Leftarrow$ assign(L, Y, X). The rules rely on additional predicates: *isConst*, *MemTag* (and, similarly, *StorageTag*, which are omitted from Fig. 8) and *Taint*. We briefly explain the meaning of these predicates and how they are derived below.

– The predicate *isConst*(O) holds if O is constant that appears in the contract. For example, the fact *isConst*$($`0`$)$ is added to the Datalog input derived for the contract in Fig. 5.

– The predicate *MemTag*(L, O, T) (and similarly *StorageTag*) defines that, at label L, the value at offset O in the memory (or storage) is assigned tag T. It is defined with three rules. The first rule encodes that writing a variable X tagged with T to a constant (i.e., known) offset O at label L, results in tagging the memory offset O at label L with tag T. The second rule defines the case when the offset is unknown, in which case all possible offsets, captured via the constant $\top$, are assigned tag T. The third rule propagates the tags to the following instructions, until reaching to an instruction that reassigns that memory location (captured by a predicate ReassignMem).

– The predicate *Taint*(L_1, L_2, X) encodes that the execution of the instruction at label L_2 depends on the value of X, where X is the condition of a goto instruction at label L_1. The first two rules defining the predicate Taint(L_1, L_2, X) taint the two branches that originate at a goto instruction at label L_1 with the condition X. Finally, the third rule propagates the tag X along the instructions of the two branches until they are merged.

MayDepOn(X, T) defines that variable X may have tag T. The first rule defines that assigning a variable X to Y results in tagging Y with X. The second rule propagates any tags of X to the assigned variable Y. The third rule propagates tags over operations with tagged variables. The three rules with mload instructions propagate tags from memory to variables. The first mload rule defines that when loading data from a constant offset O, the tags associated to that offset are propagated to the output variable Y. The second mload rules states that if the offset is unknown, then all tags of the memory are propagated to the output variable Y. Finally, the third mload rule propagates tags that are written to unknown offsets (identified by $\top$). The final rule defines that if the execution of an assign$(L, Y, _)$ instruction depends on a variable X (i.e., the label L is tainted with the variable X), then all tags assigned to X are propagated to the output variable Y.

We remark that the rules for inferring *Eq* and *DetBy* predicates are defined in a similar way and are therefore omitted.

5 SECURITY PATTERNS

In this section, we show how to express security patterns over semantics facts. We begin by defining the Securify language for expressing security patterns. Then, to define security properties formally, we provide background on the execution semantics of EVM contracts and formally define properties. We continue by presenting a set of relevant security properties, and for each, we show compliance and violation patterns, which imply the property and, respectively, its negation. This construction enables us to determine whether a contract complies with or violates a given security property. Finally, we show how Securify leverages some patterns for error-localization.

5.1 Securify Language

We first define the syntax of the language for writing patterns and then define how patterns are interpreted over the semantic facts derived for a given contract (described in Section 4).

Syntax. The syntax of the Securify language is given by the following BNF:

$$\begin{array}{rll} \varphi & ::= & \mathrm{instr}(L, Y, X, \ldots, X) \mid \mathit{Eq}(X, T) \mid \mathit{DetBy}(X, T) \\ & \mid & \mathit{MayDepOn}(X, T) \mid \mathit{MayFollow}(L, L) \mid \mathit{MustFollow}(L, L) \\ & \mid & \mathit{Follow}(L, L) \mid \exists X. \varphi \mid \exists L. \varphi \mid \exists T. \varphi \mid \neg\varphi \mid \varphi \wedge \varphi \end{array}$$

Here, L, X, and T are variables that range over program elements such as labels, contract variables, and tags. Patterns can refer to instructions $\mathrm{instr}(L, Y, X_1, \ldots, X_n)$, where instr is the instruction name, L is the instruction's label, Y is the variable storing the instruction result (if any), and $X_1, \ldots, X_n$ are variables given to the instruction as arguments (if any). Patterns can also refer to flow- and data-dependency semantic facts, which can be used to impose conditions on the labels and variables that appear in instructions. Finally, the patterns can quantify over labels, variables, and tags using the standard exists quantifier ($\exists$). More complex patterns can be written by composing simpler patterns with negation ($\neg$) and conjunction ($\wedge$).

We define several syntactic shorthands that simplify the specification of patterns. We use standard logical equivalences: we write $\forall X.\ \varphi(X)$ for $\neg(\exists X.\ \neg\varphi(X))$, $\varphi_1 \vee \varphi_2$ for $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$, and $\varphi_1 \Rightarrow \varphi_2$ for $\neg\varphi_1 \vee \varphi_2$. We also write $X = T$ for $\mathit{Eq}(X, T)$. For readability, we write: *some* $\mathrm{instr}(\overline{X}).\ \varphi(\overline{Y})$ for $\exists\overline{X}.\ \mathrm{instr}(\overline{X}) \wedge \varphi(\overline{Y})$, which imposes that there is some instruction $\mathrm{instr}(\overline{X})$ for which the logical condition $\varphi(\overline{Y})$ holds. Similarly, we write *all* $\mathrm{instr}(\overline{X}).\ \varphi(\overline{Y})$ for $\forall\overline{X}.\ \mathrm{instr}(\overline{X}) \Rightarrow \varphi(\overline{Y})$, which imposes that for all instructions $\mathrm{instr}(\overline{X})$ the condition $\varphi(\overline{Y})$ must hold.

Semantics. Patterns are interpreted by checking the inferred semantic facts:

- Quantifiers and connectors are interpreted as usual.
- Flow- and data-dependency predicates are interpreted as defined in Section 4; i.e., a semantic fact holds if and only if it is contained in the Datalog fixed-point.

For example, consider the pattern:

$$\textit{some}\ \mathrm{sstore}(L, X, Y).\ \mathit{DetBy}(X, \mathrm{caller})$$

which is a shorthand for $\exists X.\ \mathrm{sstore}(L, X, Y) \wedge \mathit{DetBy}(X, \mathrm{caller})$. This pattern is matched if there is an instruction $\mathrm{sstore}(L, X, Y)$ in the contract such that the offset X is determined by the address returned by the caller instruction (captured by the predicate $\mathit{DetBy}(X, \mathrm{caller})$). For brevity, we omit variables that are not conditioned in the pattern: *some* $\mathrm{sstore}(\_, X, \_).\ \mathit{DetBy}(X, \mathrm{caller})$.

In Fig. 9, we list security patterns that are built-in in Securify. In the following, we first give additional background on the EVM execution model and then present these patterns.

5.2 EVM Background and Properties

To understand the security properties defined in the next section, we extend the background on EVM (given in Section 4.1.1), which focused on the EVM syntax, with the semantics of EVM contracts.

EVM Semantics. A contract is a sequence of EVM instructions $C = (c_0, \ldots, c_m)$. The semantics of a contract $[\![C]\!]$ is the set of all *traces* from an initial state. A trace of a contract C is a sequence of state-instruction pairs $(\sigma_0, c_0) \rightarrow \ldots \rightarrow (\sigma_k, c_k)$, from an initial state σ_0, and such that the relation $(\sigma_j, c_j) \rightarrow (\sigma_{j+1}, c_{j+1})$ is valid according to the EVM execution semantics [52]. If a trace successfully terminates, then $c_k = \perp$. A state consists of the storage and memory state (mentioned in Section 4.1.1), stack state, transaction information, and block information. We denote by $\sigma_{S[i]}/\sigma_{M[i]}$ the value stored at offset i in the storage/memory, by σ_S/σ_M the state of the storage/memory, by σ_{Bal} the contract's balance, by σ_T the transaction, and by σ_B the block information. We denote by $t[i]$ the i^{th} pair of the trace t, for a positive i. For a negative i, $t[i]$ refers to the $i^{\text{th}} - 1$ pair of t from the end of the sequence. We denote by $\sigma^{t[i]}/c^{t[i]}$ the state/instruction of the i^{th} pair of t, and by $\sigma_f^{t[i]}$ the value of instruction f (e.g., caller) in $\sigma^{t[i]}$.

Properties. A property is a relation over sets of traces. A contract satisfies a security property ρ if $[\![C]\!] \in \rho$. If $[\![C]\!] \notin \rho$, we say that C violates the property ρ. We define relations using first-order logic formulas. The formulas are interpreted over the traces and the bitstrings that comprise the user identifiers, offsets, and other arguments or return values of the EVM instructions. We denote by $t_1, t_2, \ldots$ variables that refer to traces. We denote by $i_1, i_2, \ldots$ variables that refer to the index of a pair in a trace. We use other letters for bitstring variables. For example, we use a to refer to a bitstring which is used in the formula to refer to a user's identifier (her address), and we use x to refer to an offset in the storage or as arguments to call. For simplicity's sake, although EVM is a stack-based language, we write instructions as $r \leftarrow \mathrm{instr}(a_1, \ldots, a_k)$ and use the wildcard for arguments/return values that are not important to the formula. Note that $a_1, \ldots a_k, r$ represent the concrete values at the moment of execution.

5.3 Security Properties and Patterns

We now define seven security properties with respect to the EVM semantics [52]. Checking these properties precisely is impossible since EVM is Turing-complete. Instead, for each property, we define compliance and violation patterns over our language, which over-approximate the property and, respectively, its negation. That is, a compliance pattern match implies that the property holds, and a violation pattern match implies that the property's negation holds. If neither pattern is matched, then the property may or may not hold. In the following, for each security property, we describe its relevance, present its formal definition, and then refine it into a set of compliance and violation patterns. The complete list of properties and patterns is given in Fig. 9.

Ether Liquidity (LQ). In November 2017, a bug in a contract led to freezing \$160M [13]. The bug occurred because a contract relied on another smart contract (acting as a library) to transfer its ether to users. Unfortunately, a user accidentally removed the library contract, freezing the contract's ether. The combination of the contract being able to receive ether from users and the absence of an explicit transfer to the user led to this issue. Formally, we define this security property by requiring that *(i)* all traces t do not change the contract's balance (which means that the contract has

Property	Type	Security Pattern
LQ: Ether liquidity	*compliance*	$\mathit{all}\ \mathrm{stop}(L_1).\ \mathit{some}\ \mathrm{goto}(L_2, X, L_3).\ X = \mathrm{callvalue} \land \mathit{Follow}(L_2, L_4) \land L_3 \neq L_4 \land \mathit{MustFollow}(L_4, L_1)$
	compliance	$\mathit{some}\ \mathrm{call}(L_1, \_, \_, \mathit{Amount}).\mathit{Amount} \neq 0 \lor \mathit{DetBy}(\mathit{Amount}, \mathrm{data})$
	violation	$\big(\mathit{some}\ \mathrm{stop}(L).\ \neg \mathit{MayDepOn}(L, \mathrm{callvalue})\big) \ \land\ \big(\mathit{all}\ \mathrm{call}(\_, \_, \_, \mathit{Amount}).\ \mathit{Amount} = 0\big)$
NW: No writes after call	*compliance*	$\mathit{all}\ \mathrm{call}(L_1, \_, \_, \_).\ \mathit{all}\ \mathrm{sstore}(L_2, \_, \_).\ \neg \mathit{MayFollow}(L_1, L_2)$
	violation	$\mathit{some}\ \mathrm{call}(L_1, \_, \_, \_).\ \mathit{some}\ \mathrm{sstore}(L_2, \_, \_).\ \mathit{MustFollow}(L_1, L_2)$
RW: Restricted write	*compliance*	$\mathit{all}\ \mathrm{sstore}(\_, X, \_).\ \mathit{DetBy}(X, \mathrm{caller})$
	violation	$\mathit{some}\ \mathrm{sstore}(L_1, X, \_).\ \neg \mathit{MayDepOn}(X, \mathrm{caller}) \land \neg \mathit{MayDepOn}(L_1, \mathrm{caller})$
RT: Restricted transfer	*compliance*	$\mathit{all}\ \mathrm{call}(\_, \_, \_, \mathit{Amount}).\ \mathit{Amount} = 0$
	violation	$\mathit{some}\ \mathrm{call}(L_1, \_, \_, \mathit{Amount}).\ \mathit{DetBy}(\mathit{Amount}, \mathrm{data}) \land \neg \mathit{MayDepOn}(L_1, \mathrm{caller}) \land \neg \mathit{MayDepOn}(L_1, \mathrm{data})$
HE: Handled exception	*compliance*	$\mathit{all}\ \mathrm{call}(L_1, Y, \_, \_).\ \mathit{some}\ \mathrm{goto}(L_2, X, \_).\ \mathit{MustFollow}(L_1, L_2) \land \mathit{DetBy}(X, Y)$
	violation	$\mathit{some}\ \mathrm{call}(L_1, Y, \_, \_).\ \mathit{all}\ \mathrm{goto}(L_2, X, \_).\ \mathit{MayFollow}(L_1, L_2) \Rightarrow \neg \mathit{MayDepOn}(X, Y)$
TOD: Transaction ordering dependency	*compliance*	$\mathit{all}\ \mathrm{call}(\_, \_, \_, \mathit{Amount}).\ \neg \mathit{MayDepOn}(\mathit{Amount}, \mathrm{sload}) \land \neg \mathit{MayDepOn}(\mathit{Amount}, \mathrm{balance})$
	violation	$\mathit{some}\ \mathrm{call}(\_, \_, \_, \mathit{Amount}).\ \mathit{some}\ \mathrm{sload}(\_, Y, X).\ \mathit{some}\ \mathrm{sstore}(\_, X, \_).\ \mathit{DetBy}(\mathit{Amount}, Y) \land \mathit{isConst}(X)$
VA: Validated arguments	*compliance*	$\mathit{all}\ \mathrm{sstore}(L_1, \_, X).\ \mathit{MayDepOn}(X, \mathrm{arg}) \Rightarrow \big(\mathit{some}\ \mathrm{goto}(L_2, Y, \_).\ \mathit{MustFollow}(L_2, L_1) \land \mathit{DetBy}(Y, \mathrm{arg})\big)$
	violation	$\mathit{some}\ \mathrm{sstore}(L_1, \_, X).\ \mathit{DetBy}(X, \mathrm{arg}) \Rightarrow \neg\big(\mathit{some}\ \mathrm{goto}(L_2, Y, \_).\ \mathit{MayFollow}(L_2, L_1) \land \mathit{MayDepOn}(Y, \mathrm{arg})\big)$

Figure 9: Compliance and violation security patterns for relevant security properties

no ether and thus its ether is vacuously liquid), or *(ii)* there exists a trace t that decreases the contract's balance (i.e., ether is liquid).

$$\psi_{LQ} = (\forall t. \sigma^{t[0]}_{Bal} = \sigma^{t[-1]}_{Bal}) \ \lor (\exists t. \sigma^{t[0]}_{Bal} > \sigma^{t[-1]}_{Bal})$$

To over-approximate ψ_{LQ} with our language, we leverage the fact that if ether is transferred to the contract, then the amount of ether transferred is given by the callvalue instruction. Thus, if, for all traces that complete successfully, this amount is zero, then the first part of ψ_{LQ} is satisfied. These is exactly the first liquidity compliance pattern in Fig. 9: it matches if all transactions that can complete successfully (reach a stop instruction) have to follow a branch of a condition (where the condition is identified by a goto instruction) that is reachable only if the ether transferred to this contract is zero (this branch is the one to which the goto instruction does not jump). The second liquidity compliance pattern over-approximates the second part of ψ_{LQ}. It leverages the fact that ether is liquid if there is a reachable call instruction which sends a non-zero amount of ether. Concretely, it is matched if there is a call instruction which transfers *(i)* a positive amount of ether or *(ii)* amount of ether which depends only on the transaction data, and thus can be positive.

Our violation pattern over-approximates $\neg\psi_{LQ}$ by checking that both conditions are false: the contract can receive ether, but cannot transfer ether. To guarantee that the contract can receive ether, it verifies that there is an execution that can complete successfully (i.e., reach stop) and its execution does not depend on callvalue – this guarantees that some trace with positive callvalue can complete. To guarantee that ether cannot be transferred, it verifies that all call instructions transfer 0 ether.

No Writes After Calls (NW). In July 2016, a bug in the DAO contract enabled an attacker to steal \$60M [1]. The attacker exploited the combination of two factors. First, a call instruction which upon its execution enabled the recipient of that call to execute her own code before returning to the contract. Second, the amount transferred by this call depended on a storage value, which was updated *after* this call. This value was critical as it recorded the amount of ether that the call's recipient had in the contract, and can thus request to receive. This allowed the attacker to call the function again *before the storage was updated*, thus making the contract believe that the user still had ether in the contract. A property that captures when this attack cannot occur checks that there are no writes to the storage after any call instruction. We formalize this vulnerability by requiring that, for all traces t, the storage does not change in the interval that starts just before any call instruction and ends when the trace completes:

$$\psi_{NW} = \forall t \forall i (i < -1 \land c^{t[i]} = \_ \leftarrow \mathrm{call}(\_, \_, \_)) \Rightarrow \sigma^{t[i]}_S = \sigma^{t[-1]}_S$$

Note that this property is different from reentrancy [39], which stipulates that the callee must not be able to re-enter the same function and reach the call instruction. Our compliance rule over-approximates ψ_{NW} by leveraging the fact that the storage can only be changed via sstore. It is thus matched if call instructions are not followed by sstore instructions. Our violation pattern over-approximates $\neg\psi_{NW}$ by checking that there is a call instruction which must be followed by a write to the storage, in which case the implication of ψ_{NW} is violated.

Restricted Writes (RW). In July 2017, an attacker stole $30M because of an unrestricted write to the storage [10]. The attacker exploited the reliance of the contract on a library that enabled to unconditionally set an *owner* field to any address. This enabled the attacker to take ownership over the contract and steal its ether. We consider a security property that guarantees that writes to storage are restricted. The property requires that, for every storage offset x (e.g., a field in the contract), there is a user a that cannot write at offset x of the storage.

$$\psi_{RW} = \forall x \exists a \forall t (\sigma^{t[0]}_{\text{caller}} = a \Rightarrow c^{t[-1]} \neq \text{sstore}(\_, x, \_))$$

Our compliance pattern over-approximates ψ_{RW} by checking that offsets of sstore instructions, denoted x, are determined by the sender's identifier (i.e., users can only write to their designated slot). This ensures that for all x, there exists a user a (in fact, all users but one) who cannot write to x. The violation pattern over-approximates $\neg\varphi_{RW}$ by checking if there is an sstore instruction whose execution and offset are independent of caller. In this case, we can define an offset x, for which all users can write – hence violating the property. If this property is too restrictive (there are cases where it is safe to allow global writes to the storage), one can define it (and adapt the patterns) with respect to critical writes (e.g., writes that modify an `owner` field), identified by their label l.

In the following, we skip the formal definition of properties, and only describe them informally.

Restricted Transfer (RT). We define a property that guarantees that ether transfers (via call) cannot be invoked by any user a. Violation of this property can detect Ponzi schemes [20]. Our compliance pattern requires that for all users, invocations of that call instruction do not transfer ether. Our first violation pattern checks if the call instruction transfers non-zero amount of ether and its execution is independent of the sender. For the second violation pattern, the amount of ether transferred depends on the transaction data (and thus can be set to a non-zero value), while the execution is independent of this data (and will thus take place).

Handled Exception (HE). In February 2016, a contract by the name "King of Ether" had an issue due to mishandled exceptions, forcing its creator to publicly ask users not to send ether to it [4]. The issue was that the return value of a call, which indicated if the instruction completed successfully, was not checked.

Our compliance pattern checks that call instructions are followed by a goto instruction whose condition is determined by the return code of call. This guarantees that depending on the return code, different execution paths are taken. Our violation pattern checks that the call instruction is not followed by a goto instruction which may depend on the return value. This guarantees that there is no different behavior depending on the result of the call.

Transaction Ordering Dependency (TOD). An inherent issue in the blockchain model is that there is no guarantee on the execution order of transactions. While this has been known, it recently became critical in the context of Initial Coin Offerings, a popular means for start-ups to collect money by selling tokens. The initial tokens are sold at a low price while offering a high bonus, and as demand increases the price increases and the bonus decreases. It has been observed that miners exploit this to create their transactions to win the big bonus at a low rate [14].

Our compliance pattern requires that the amount of ether send by a call instruction is independent of the state of the storage and contract's balance. This means that reordering transactions (which can be affected by changing the storage or balance) does not affect the amount sent by the call execution. Our violation pattern checks that the amount of the call instruction is determined by a value read from the storage, whose offset in the storage is known (i.e., it is constant), and that this value can be updated.

In Section 7, we evaluate several versions of the TOD property: *(i) TOD Transfer (TT)* indicates that the execution of the ether transfer depends on transaction ordering (e.g., a condition guarding the transfer depends on the transaction ordering); *(ii) TOD Amount (TA)* marks that the amount of ether transferred depends on the transaction ordering (this variation is the one described above and in Fig. 9); *(iii) TOD Receiver (TR)* captures the vulnerability that the recipient of the ether transfer might change, depending on the transaction ordering.

Validated Arguments (VA). Method arguments should be validated before usage, because unexpected arguments may result in insecure contract behaviors. Contracts must check whether all transaction arguments meet their desired preconditions.

Our compliance pattern checks that before storing in the persistent memory a variable that may depend on a method argument, there exists a check of the argument value. Our violation pattern identifies sstore instructions that write to memory a method argument without previously checking its value.

Limitations. We next discuss a few limitations of checking properties through patterns. First, all our violation patterns assume that the violating instructions (which match the violation pattern) are part of some terminating execution. For example, in the violation pattern of ether liquidity, the matching stop is assumed to be reachable, and in the violation pattern of no writes after calls, both the call and the write are assumed to be part of some terminating execution. We take this assumption since, in general, this problem is undecidable.

Second, the security properties we consider are generic and do not capture contract-specific requirements (we illustrate the specification of contract-specific patterns in Securify's DSL below). Some vulnerabilities are, however, contract-specific, and therefore they are not captured by our compliance patterns (i.e., a contract can be exploitable even if a compliance pattern is matched). For example, our compliance pattern for handled exceptions matches if there is *some* check over the call's return value. However, the pattern cannot check that the exception was handled *correctly*, as this is contract-specific. Similarly, the compliance pattern for validated arguments matches if there is *some* check over the arguments. However, the check can still miss cases where inputs are not correctly validated, as the meaning of *correctly validated* varies across contracts.

Third, since our patterns do not capture precisely their corresponding properties, it can happen that a contract matches neither the compliance nor the violation pattern. In this case, Securify cannot infer whether the property holds, and thus shows a warning.

Contract-specific Patterns. Finally, we remark that Securify is not limited to checking the security properties described above. In fact, it is common that a security auditor would write custom patterns defined for a particular contract. Such custom patterns are specified by providing an expression in the Securify language.

To illustrate this, suppose an auditor wants to check whether the execution of a specific sensitive call instruction at label l depends on the address of the owner. To discover violations of this property, the auditor would write:

$$\textit{some}\ \text{call}(L, \_, \_, \_).\\ \quad (L = l) \land \neg(\textit{some}\ \text{sload}(\_, \textit{Owner}, X).\ \textit{MayDepOn}(L, X))$$

Here, *Owner* is the identifier of the field storing the owner address, i.e. a constant offset in the contract's storage.

5.4 Error Localization via Violation Patterns

An important part of Securify is to pinpoint the instructions that lead to violations (or potential violations) of security properties, as this enables developers to fix the code. In this section, we characterize which patterns enable such error localization. We call such patterns *instruction patterns* (as they pinpoint instructions), and we call other patterns *contract pattern* (as the violation is identified for the entire contract).

Instruction Patterns. An instruction pattern has the form of: $\textit{some}\ \text{instr}(\overline{X}).\ \varphi_v(\overline{X})$, for violation patterns, and, $\textit{all}\ \text{instr}(\overline{X}).\ \varphi_c(\overline{X})$, for compliance patterns. That is, if a violation pattern is an instruction pattern and it is matched by some $\text{instr}(\overline{X})$, then Securify can highlight this instruction as a violation. Similarly, if a compliance pattern is an instruction pattern and it is *not* matched because of some $\text{instr}(\overline{X})$, then Securify can highlight this instruction as a warning (assuming that the corresponding violation pattern has not matched). Note that six of the violation patterns in Fig. 9 (all except the violation pattern for ether liquidity) are instruction patterns.

Contract Patterns. Patterns which are not instruction patterns are called *contract patterns*. For them, it is difficult to pinpoint a single instruction responsible for its violation. The ether liquidity violation pattern is an example of a contract pattern: it conjoins two different conditions pertaining to stop and call instructions. For contract patterns, Securify evaluates the compliance and violation patterns and flags the contract as vulnerable (if the violation pattern is matched) or issues a warning (if no pattern is match) without pinpointing specific instructions.

6 IMPLEMENTATION

In this section, we detail the implementation of Securify.

Decompiler. The decompiler transforms the EVM bytecode provided as input into the corresponding assembly instructions, as defined in [52]. Next, it converts the EVM instructions into an SSA form. The SSA instructions are identical to the EVM instruction set except that they exclude stack operations (e.g., pop, push, etc.). Our conversion method is similar to the one described in [45, 51]. The decompiler constructs the control flow graph (CFG) on top of the decompiled instructions.

	EVM dataset	Solidity dataset
# Contracts	24, 594	100
# call instructions	46, 106	67
# sstore instructions	56, 346	297

Figure 10: Statistics of the two Ethereum datasets

Optimizations. Securify employs three optimizations over the CFG, which improve the precision of its analysis:

(i) *Unused instructions*, which eliminates any instructions whose results are not used. On average, this optimization reduces the contract's instructions by 44% and improves the scalability and precision of the subsequent analysis.

(ii) *Partial evaluation*, which propagates constant values along computations [29]. This step improves the precision of storage and memory analysis (e.g., *MemTag*). As we show in our evaluation, partial evaluation resolves over 70% of the offsets that appear in storage/memory instructions.

(iii) *Method inlining*, which improves the precision of the static analysis by making it context sensitive.

Inference of Semantic Facts. Securify derives semantic facts using inference rules specified in stratified Datalog, using the Soufle Datalog solver [36] to efficiently compute a fixed-point of all facts. We report on concrete numbers in Section 7.

Evaluating Patterns. To check the security patterns, Securify iterates over the instructions to handle the *all* and *some* quantifiers in the patterns. Then, to check inferred facts, it directly queries the fixed-point computed by the Datalog solver. If a violation pattern is matched, Securify reports which instructions are identified as vulnerable, to provide error-localization for users. If no pattern is matched, Securify reports a warning, to indicate that an instruction may or may not be vulnerable.

7 EVALUATION

To evaluate Securify, we conducted the following experiments: *(i)* evaluated Securify's effectiveness in proving the correctness of and discovering violations in real-world contracts; *(ii)* manually inspected Securify's results (i.e., reported violations and warnings) on smart contracts whose source code had been uploaded to Securify's public interface; *(iii)* compared Securify to Oyente [39] and Mythril [16], two smart contract checkers based on symbolic execution; *(iv)* measured the success of Securify's decompiler in resolving memory and storage offsets; *(v)* measured Securify's time and memory consumption.

Datasets. We used two datasets of smart contracts to evaluate Securify. Our first dataset, dubbed *EVM dataset*, consists of 24, 594 smart contracts obtained by parsing create transactions using the parity client [12]. Using create transactions, we obtained the EVM bytecode of these smart contracts. Our second dataset, dubbed *Solidity dataset*, consists of 100 smart contracts written in Solidity which were uploaded to Securify's public interface. To avoid bias, we selected the first 100 contracts in alphabetical order uploaded in 2018. To simplify manual inspection, we restricted our selection to contracts with up to 200 lines of Solidity code.

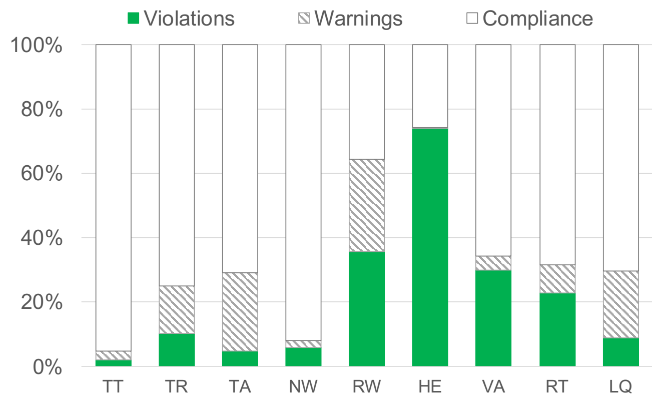


Figure 11: SECURIFY results on the EVM dataset. The violations and compliance segments indicate instructions that are proved to be safe/violations for each security property.

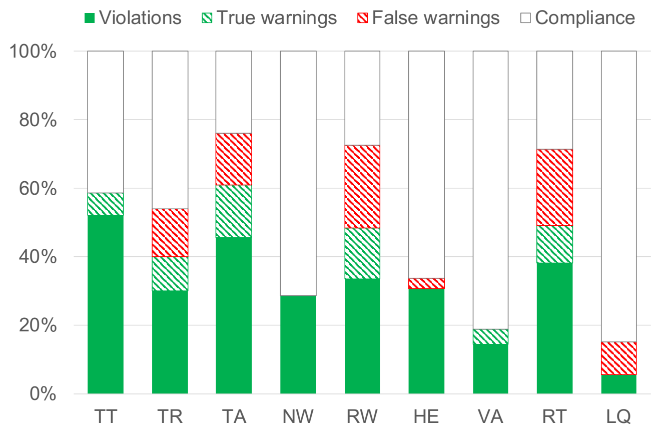


Figure 12: SECURIFY results on the Solidity dataset. The warnings are classified into true and false warnings based on whether they indicate a security issue or not.

We give relevant statistics on the two datasets in Fig. 10. Note that the number of contracts defines the relevant checks for the ether liquidity (LQ) property, the number of sstore instructions defines the relevant instructions for the restricted writes (RW) and the validated arguments (VA) property, and the number of call instructions defines the relevant instructions for the remaining properties.

Security Analysis of Real-World Smart Contracts. In this task, we evaluate SECURIFY's effectiveness in proving security properties (i.e., matching a compliance pattern) and finding violations (i.e., matching a violation pattern) in real-world contracts. To this end, we ran SECURIFY on all smart contracts contained in our EVM dataset and measured the fraction of violations, warnings, and compliances reported by SECURIFY.

Fig. 11 summarizes the results. The figure shows one bar for each security property. Each bar has three segments: *(i) violations*, which shows the fraction of instructions that have matched a violation pattern of the given property, *(ii) warnings*, which shows the fraction of instructions that have not matched any pattern (neither violation or compliance pattern) of the given property, and *(iii) compliance*, which shows the fraction of instructions that have matched a compliance pattern of the given property. We note that the sum of the three segments adds up to 100%.

For example, consider the no writes after calls (NW) property. The data shows that 6.5% of the call instructions violate the property, 90.9% are proved to be compliant, and the remaining 2.6% are reported as warnings. On average across all security properties, SECURIFY successfully proves that 55.5% of the relevant instructions are safe, 29.3% are definite violations, and it reports 15.2% warnings. Further, 65.9% of all instructions that failed to match a compliance pattern (and hence may indicate an error) are successfully proved to be definite violations (using the violation patterns). This indicates a reduction of 65.9% in the number of instructions that users must manually classify into true warnings and false warnings. We report on the precise breakdown between false and true warnings in our next experiment.

Overall, our results indicate that SECURIFY's compliance and violations patterns are expressive enough to prove and, respectively, disprove relevant security properties. Further, we note that since SECURIFY is extensible, one can further refine SECURIFY's results by extending it with additional patterns that would convert more warnings into violations and compliances. This would benefit some of the security properties that are harder to prove or disprove (such as restricted writes).

Manual Inspection of Results. In our second experiment, we manually inspected SECURIFY's reports to gain a better understanding of its results. To this end, we ran SECURIFY on all contracts contained in our Solidity dataset. We then manually classified each reported warning as a *true warning* if it indicates a violation of the security property, and otherwise, we classified it as a *false warning*. We also inspected and confirmed the correctness of all reported violations and compliances.

Fig. 12 summarizes our results. As before, the figure shows one bar for each security property. In addition to the violation and compliance segments, we partition the segment with reported warnings into *true warnings* and *false warnings*.

Consider the handled exception (HE) property. The data shows that SECURIFY successfully proves that 29.9% of the call instructions have return values that are *not* checked by the code (indicating a violation of the property). Further, SECURIFY proves that these error values *are* checked for the remaining 70.1% of call instructions. SECURIFY does not issue any warnings for this property because it matched at least on of the patterns for each of the call instructions.

We remark that the number of security issues discovered in the Solidity dataset is higher relative to those found in the EVM dataset. We believe this is due to the fact that the two datasets come from different distributions: the Solidity dataset consists of recent contracts (uploaded in 2018) that are still in development stage. In contrast, the EVM dataset contains all contracts deployed on the blockchain. Further, users often deliberately uploaded vulnerable contracts to experiment and evaluate SECURIFY. An exception is the reduction in handled exception property (HE), which has more violations in

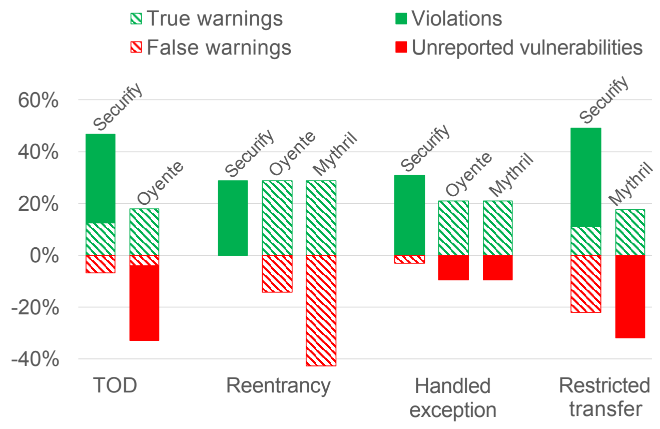


Figure 13: Comparing SECURIFY to Oyente and Mythril

the EVM dataset compared to the Solidity dataset. We believe this is due to the fact that developers now use the `transfer()` statement for ether transfers, which handles errors by default and was specifically introduced to avoid issues due unhandled exceptions.

We observe that the effectiveness of the patterns varies across properties, which is expected as some properties are more difficult to prove/disprove than others. For example, the restricted transfer property (RT) and the three transaction ordering dependence properties (TT, TR, and TA) are hard to prove correct and result in a relatively high number of false warnings (roughly half of the warnings are false warnings). However, for other security properties, such as no writes after calls (NW) and handled exception (HE), all warnings issued by SECURIFY indicate true warnings, indicating that the corresponding compliance patterns precisely matches contracts that satisfy these properties.

Comparing SECURIFY to Symbolic Security Checkers. We now compare SECURIFY to two recent open-source security checkers based on symbolic execution – Oyente [39] and Mythril [16]. To compare the three systems, we ran the latest versions of Oyente and Mythril against all contracts in our Solidity dataset, for which we have already manually classified all warnings into true and false warnings. Oyente supports three of SECURIFY's security properties: TOD, which checks the disjunction of the TOD receiver and TOD amount properties, reentrancy (called no writes after calls[1] in SECURIFY), and handled exceptions. Mythril also supports the reentrancy and handled exception properties, and in addition, implements a check of the restricted transfer property.

Our results are summarized in Fig. 13. For SECURIFY, we report the fraction of reported violations, true warnings, and false warnings. Since both Oyente and Mythril may report false positives (Oyente has false positives because their checks do not imply a contract vulnerability, as shown in [30]), we treat all bugs listed by them as *warnings* as they must be classified by the user into true warnings and false warnings. Note that, unlike SECURIFY, Oyente and Mythril do not report definite violations, i.e., results that are guaranteed to violate security properties. Since Oyente and Mythril explore a subset of all contract's behaviors, they may fail to report certain vulnerabilities, and we report these as *unreported vulnerabilities* in the figure. We depict true warnings and violations above the X-axis (to indicate desirable results), and we plot false warnings and unreported vulnerabilities below the X-axis (to indicate undesirable results).

We observe that for all properties except reentrancy, Oyente and Mythril miss to report some actual vulnerabilities. Oyente fails to report 72.9% of TOD violations, and Mythril fails to report 65.6% of the restricted transfer violations. Overall, the two symbolic tools fail to report vulnerabilities for all considered security properties.

[1]We remark that to ensure the absence of storage writes after call instructions, Oyente checks that the user cannot re-enter and reach the same call instruction.

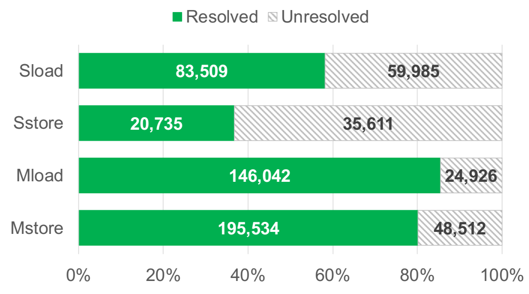


Figure 14: Offsets resolved by partial evaluation.

Resolving Storage/Memory Offsets. We report on SECURIFY's partial evaluation optimization for resolving memory and storage offsets. Fig. 14 shows the total number of mload, mstore, sload and sstore instructions found in our EVM dataset. The figure depicts the number of resolved offsets. On average across all four instructions, partial evaluation correctly resolves 72.6% of the offsets. This indicates that SECURIFY can often infer the precise writes to storage/memory, thereby improving the precision of the subsequent analysis. Memory offsets are more often resolved than storage offsets, as the latter often depend on user-provided inputs.

Time and Memory Consumption. SECURIFY terminates for all contracts and takes on average 30 seconds per contract (to check all compliance and violation patterns). Oyente and Mythril have similar running times when used with default settings (which do not provide full coverage). To improve the coverage of these tools, users must increase the constraint solving timeouts and loop bounds, which in turn result in increased running times (especially for larger contracts). The memory consumption of SECURIFY is determined by the size of the fixed point analysis. In 95% of cases, the consumption was below 10MB, and in the rest it was below 1GB.

Summary. Overall, our results indicate that SECURIFY's patterns are effective in finding violation and establishing correctness of contracts. Going further, we see two relevant items for future work. First, it would be interesting to integrate SECURIFY with existing frameworks that provide formal EVM semantics, such as [30, 32], as a way to further validate SECURIFY's analysis and patterns, and to formally prove the guarantees it provides. Second, we can leverage SECURIFY to improve existing symbolic checkers, such as Oyente and Mythril. For example, SECURIFY's compliance patterns can be used to reduce the false positive rate of these tools.

8 RELATED WORK

We discuss some of the works that are most closely related to ours.

Analysis of Smart Contracts. Smart contracts have been shown to be exposed to severe vulnerabilities [19, 25]. Hirai [33] was one of the firsts to formally verify smart contracts using the Isabelle proof assistant. In [34], Hirai defines a formal model for the Ethereum Virtual Machine using the Lem language. This model proves safety properties of smart contracts using existing interactive theorem provers. Formal semantics of the EVM have been defined by Grishchenko *et al.* [30] using the F* framework and by Hildenbrandt *et al.* [32] using the K framework [46]. These semantics are executable and were validated against the official Ethereum test suite. Further, they enable the formal specification and verification of properties. The main benefit of these frameworks is that they provide strong formal verification guarantees and are precise (no false positives). They target arbitrary properties, but are, unfortunately, nontrivial to fully automate. In contrast, Securify targets properties that can be proved/disproved by checking simpler properties that can be verified in a fully automated way.

In the space of automated security tools for smart contracts, there are several popular systems based on symbolic execution. Examples include Oyente [39], Mythril [16], and Maian [44]. While symbolic execution is indeed a powerful generic technique for discovering bugs, it does not guarantee to explore all program paths (resulting in false negatives). In contrast to these tools, Securify explores all contract behaviors. In the context of smart contracts, path constraints often involve hard-to-solve constraints, such as hash-functions, resulting in low coverage or false positives. Further, to avoid false positives, symbolic tools must precisely explore the set of feasible contract blocks. Towards this, Maian already uses a concrete validation step to filter false positives. An interesting application of Securify would be to filter the false positives reported by symbolic tools using its compliance patterns. In contrast to the approaches based on symbolic execution, Securify is an abstract interpreter. As such, it can provide soundness guarantees over all possible executions. This is different from symbolic execution which can only guarantee soundness if the number of paths can be bounded (in particular, this means that loops have to be unrolled). Even when the number of paths is bound, an abstract interpreter often scales better than symbolic execution since it can join paths and does not have to explore different paths separately. On the other hand, symbolic execution can, in principle, handle more expressive predicates (within the logic of the underlying SMT solver), and, in theory, it has no false positives (in practice, as we show in Fig. 13, it can have false positives).

Bhargavan *et al.* [21] present preliminary work on verifying Ethereum smart contracts by translating Solidity and EVM bytecode to an existing verification system. The paper does not report how their tool performs on real-world contracts. The work presented in [22] combines game theory and probabilistic model checking to validate a decentralized smart contract protocol.

The Zeus system [37] is a sound analyzer that translates smart contracts to the LLVM framework. Zeus uses XACML as a language to write properties. In contrast, Securify's DSL supports the checking of data- and control-flow properties. Further, Zeus does not support violation patterns as a way to reduce false positives. We could not directly compare Securify with Zeus as neither Zeus nor its benchmarks are publicly available.

Similarly to Securify, the work by Grossman *et al.* [31] also targets domains-specific properties. In more detail, they introduce a dynamic linearizability checker to identify reentrancy issues. In contrast, Securify supports a larger class of properties for smart contracts and supports a DSL to allow security experts to extend the system with more properties.

Security Factors. Delmolino *et al.* [28] document the kinds of vulnerabilities students introduce while writing smart contracts and propose methods on how to avoid common pitfalls. Chen *et al.* [27] show that the current standard compiler Solidity does not properly optimize the EVM bytecode. Seijas *et al.* [47] overview the capabilities of different blockchains such as Bitcoin, Nxt, and Ethereum, and survey extensions (Kosba *et al.* [38]).

Language-Based Security. Programming language approaches enforce security at the program code level. PQL [42] introduces a program query language for Java that allows developers to express patterns of interest and check Java programs against them. Both [42] and our work have an underlying declarative solver for the static analysis. Pidgin [35] is a custom query language for program dependence graphs that can also capture security properties on Java programs. In contrast, our work focuses on Ethereum smart contracts. Securify's analysis is tailored to the Ethereum setting, such as Ethereum-specific instructions (e.g., balance) and reasoning across memory and contract storage. Furthermore, Securify provides a DSL specific to security properties for smart contracts.

Declarative Program Analysis. Declarative approaches to program analysis are related to Securify's fact inference engine, as they also rely on Datalog to express analysis computations. The Doop framework [23, 49] presents a fast and scalable declarative points-to analysis for Java programs and is one of the first works to show the promise of declarative static analysis. Following these ideas, the authors of [53] present a technique for automatic abstraction refinement for static analysis specified in Datalog, and in [41] the authors propose to involve the developer in the abstraction-refinement loop. Researchers have developed specific extensions to Datalog, such as Flix [40], to improve the efficiency and scalability of Datalog-based program analysis. These works are orthogonal to Securify's inference engine. They develop general program analysis techniques, while Securify leverages these advances for reasoning about smart contracts. As such, Securify can benefit from any future advances in Datalog-based program analysis.

9 CONCLUSION

We presented Securify, a new lightweight and scalable verifier for Ethereum smart contracts. Securify leverages the domain-specific insight that violations of many practical properties for smart contracts also violate simpler properties, which are significantly easier to check in a purely automated way. Based on this insight, we devised compliance and violation patterns that can effectively prove whether real-world contracts are safe/unsafe with respect to relevant properties. Overall, Securify enjoys several important benefits: *(i)* it analyzes all contract behaviors to avoid undesirable false negatives; *(ii)* it reduces the user effort in classifying warnings into true positives and false alarms by guaranteeing that certain behaviors are actual errors; *(iii)* it supports a new domain-specific language that enables users to express new vulnerability patterns as they emerge; finally, *(iv)* its analysis pipeline – from bytecode decompilation, optimizations, to checking of patterns – is fully automated using scalable, off-the-shelf Datalog solvers.

ACKNOWLEDGMENTS

The research leading to these results was partially supported by an ERC Starting Grant 680358. We thank Hubert Ritzdorf and the ChainSecurity team for their valuable contributions to this project.

REFERENCES

[1] 2016. The DAO Attacked: Code Issue Leads to 60 Million Ether Theft. (2016).
[2] 2016. Etherdice. (2016). Available from: https://etherdice.io/.
[3] 2016. King of Ether. (2016). Available from: https://github.com/kieranelby/KingOfTheEtherThrone/blob/v0.4.0/contracts/KingOfTheEtherThrone.sol.
[4] 2016. King of Ether, Postmortem. (2016). Available from: https://www.kingoftheether.com/postmortem.html.
[5] 2016. Reentrancy Woes in Smart Contracts. (2016). Available from: http://hackingdistributed.com/2016/07/13/reentrancy-woes/.
[6] 2016. theDAO. (2016). Available from: https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413.
[7] 2017. Accidental bug may have frozen $280 million worth of digital coin ether in a cryptocurrency wallet. (2017). Available from: https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen...
[8] 2017. Blockchain is empowering the future of insurance. (2017). Available from: https://techcrunch.com/2016/10/29/blockchain-is-empowering-the-future-of-insurance/.
[9] 2017. ETHLance. (2017). Available from: http://ethlance.com/.
[10] 2017. An In-Depth Look at the Parity Multisig Bug. (2017). Available from: http://hackxingdistributed.com/2017/07/22/deep-dive-parity-bug.
[11] 2017. Northern Trust uses blockchain for private equity record-keeping. (2017). Available from: http://www.reuters.com/article/nthern-trust-ibm-blockchain-idUSL1N1G61TX.
[12] 2017. Parity Ethereum Client. (2017). Available from: https://github.com/paritytech/parity.
[13] 2017. Security Alert. (2017). Available from: https://paritytech.io/blog/security-alert.html.
[14] 2017. Submarine Sends: IC3's Plan to Clamp Down on ICO Cheats. (2017). Available from: https://www.coindesk.com/submarine-sends-inside-ic3s-plan-to-clamp-...
[15] 2018. Ethereum Smart Contract Security Best Practices. (2018). Available from: https://consensys.github.io/smart-contract-best-practices/.
[16] 2018. Mythril. (2018). Available from: https://github.com/ConsenSys/mythril.
[17] 2018. Parity Wallet Library. (2018). Available from: https://github.com/paritytech/parity/blob/4d08e7b0aec46443bf26547b17d10cb302672835/js/src/contracts/snippets/enhanced-wallet.sol.
[18] 2018. Solidity, high-level language for writing smart contracts. (2018). Available from: https://solidity.readthedocs.io/en/develop/.
[19] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust - 6th International Conference, POST*. 164–186.
[20] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. 2017. Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact. *CoRR* abs/1703.03779 (2017).
[21] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS)*. 91–96.
[22] Giancarlo Bigi, Andrea Bracciali, Giovanni Meacci, and Emilio Tuosto. 2015. Validation of Decentralised Smart Contracts Through Game Theory and Formal Methods. In *Programming Languages with Applications to Biology and Security*. 142–161.
[23] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 243–262.
[24] Vitalik Buterin. 2013. Ethereum: a next generation smart contract and decentralized application platform. (2013). Available from: https://github.com/ethereum/wiki/wiki/White-Paper.
[25] Vitalik Buterin. 2016. Thinking About Smart Contract Security. (2016). Available from: https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/.
[26] Pawel Bylica. 2017. How to Find $10M Just by Reading the Blockchain. (Apr 2017). Available from: https://blog.golemproject.net/how-to-find-10m-by-just-reading-blockchain-6ae9d39fcd95.
[27] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *Software Analysis, Evolution and Reengineering (SANER)*. 442–446.
[28] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. 2016. Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. In *Financial Cryptography and Data Security (FC)*. 79–94.
[29] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391.
[30] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust - 7th International Conference (POST)*. 243–269.
[31] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL* 2, POPL (2018), 48:1–48:28.
[32] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *31st IEEE Computer Security Foundations Symposium (CSF)*. 204–217.
[33] Yoichi Hirai. 2016. *Formal verification of Deed contract in Ethereum name service.* Technical Report. Available from: https://yoichihirai.com/deed.pdf.
[34] Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Financial Cryptography and Data Security (FC)*. 520–535.
[35] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. 2015. Exploring and Enforcing Security Guarantees via Program Dependence Graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 291–302.
[36] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification - 28th International Conference (CAV)*. 422–430.
[37] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium (NDSS)*.
[38] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *IEEE Symposium on Security and Privacy (SP)*. 839–858.
[39] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 254–269.
[40] Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to flix: a declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 194–208.
[41] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. 2015. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 462–473.
[42] Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. 2005. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 365–383.
[43] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).
[44] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. *CoRR* abs/1802.06038 (2018).
[45] Todd A. Proebsting and Scott A. Watterson. 1997. Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?). In *Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*. 185–198.
[46] Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.
[47] Pablo Lamela Seijas, Simon Thompson, and Darryl McAdams. 2016. Scripting smart contracts for distributed ledger technology. *IACR Cryptology ePrint Archive* 2016 (2016).
[48] Gagandeep Singh, Markus Püschel, and Martin Vechev. 2017. Fast Polyhedra Abstract Domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 46–59.
[49] Yannis Smaragdakis and Martin Bravenboer. 2010. Using Datalog for Fast and Easy Program Analysis. In *Datalog Reloaded - First International Workshop, Datalog*. 245–251.

[50] Jeffrey D. Ullman. 1988. *Principles of Database and Knowledge-base Systems, Vol. I.* Principles of computer science series, Vol. 14.

[51] Raja Vallee-Rai and Laurie J. Hendren. 1998. Jimple: Simplifying Java Bytecode for Analyses and Transformations. (1998).

[52] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* (2014).

[53] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in Datalog. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*. 239–248.