sVerify: Verifying Smart Contracts Through Lazy Annotation and Learning

Bo Gao[1(✉)], Ling Shi[2], Jiaying Li[2], Jialiang Chang[3], Jun Sun[2], and Zijiang Yang[3]

[1] Singapore University of Technology and Design, Singapore, Singapore
[2] Singapore Management University, Singapore, Singapore
[3] Western Michigan University, Kalamazoo, USA

Abstract. Smart contracts have recently attracted much attention from industry as they aim to assure anonymous distributed secure transactions. It also becomes clear that they are not immune to code vulnerabilities. As smart contracts cannot be patched once deployed, it is crucial to verify their correctness before deployment. Existing approaches mainly focus on testing and bounded verification which do not guarantee the correctness of smart contracts. In this work, we develop a formal verifier called *sVerify* for Solidity smart contracts based on a combination of lazy annotation and automatic loop invariant learning techniques. The latter is essential as explicit or implicit loops (due to fallback function calls) are common in smart contracts. Patterns and features which are specific to smart contracts are used to facilitate invariant learning. *sVerify* has been evaluated with 4670 Solidity smart contracts, and the evaluation result shows that *sVerify* is effective and reasonably efficient for verifying smart contracts.

Keywords: Verification · Smart contracts · Loop invariant learning

1 Introduction

Blockchain is a fast-growing research area in recent years. It is first conceptualized in Bitcoin blockchain [23] by Satoshi Nakamoto based on multiple techniques like cryptographic chain of blocks by Stuart Haber and W. Scott Stornetta [12], distributed systems by Lamport [16], etc. The emergence of Bitcoin makes financial transactions among strangers possible without the help of a third-party authority. Later on, Buterin stepped forward to develop the platform Ethereum [29], which allows self-enforcing programs, called smart contracts, to run by themselves. Smart contracts have since attracted much attention in many domains, such as financial institutes and supply chains.

A smart contract is a computerized transaction protocol that executes the terms of a contract to satisfy user requirements, such as voting and trading. It can be regarded as a computer program, which is typically written in a Turing-complete language called Solidity in Ethereum. The immutability of blockchain makes smart contracts unpatchable once they are deployed on the blockchain. Furthermore, the Javascript-like syntax of Solidity and its many unique language features (e.g., storage variables and fallback functions) often confuse users, even if they are experienced with traditional programming languages. As a result, there are many attacks due to code vulnerabilities that caused huge economic losses. For instance, the DAO attack [1] resulted in a loss roughly

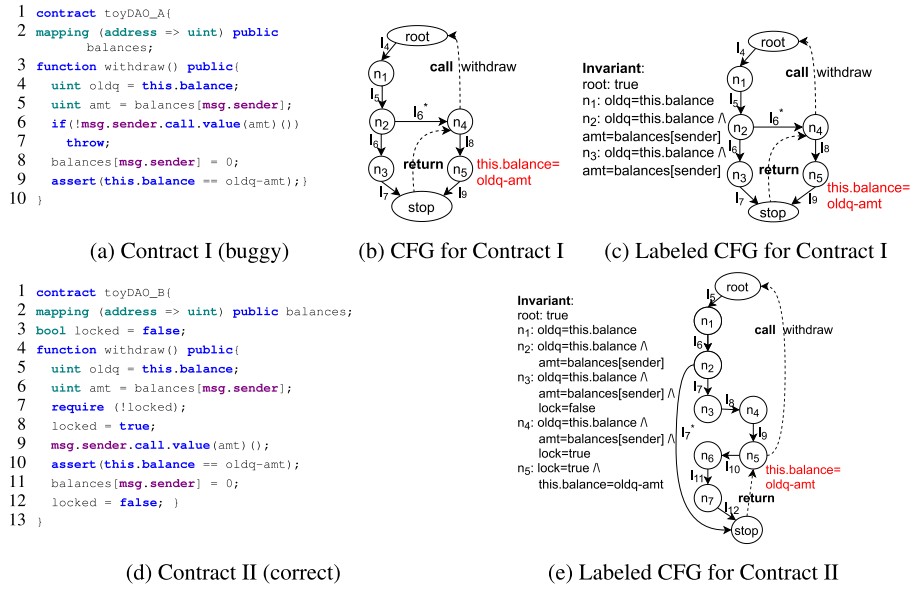
T. Margaria and B. Steffen (Eds.): ISoLA 2021, LNCS 13036, pp. 453–469, 2021.
https://doi.org/10.1007/978-3-030-89159-6_28

equivalent to 60 million USD at the time. The attacker found a loophole in the *splitDAO* function so that he could repeatedly withdraw Ether through an implicit loop in the *fallback* function in a single transaction.

To react on the increasing amount of attacks on smart contracts, multiple approaches and tools have been developed to analyze the correctness in recent years. For instance, Luu *et al.* [20] developed a symbolic engine for Solidity smart contracts called Oyente, which systematically analyzes individual functions in a smart contract to identify vulnerabilities. Nikolic *et al.* [24] developed a symbolic analyzer called MAIAN, which performs inter-procedural symbolic analysis to check suicidal, prodigal, and greedy contracts based on the bytecode of Ethereum smart contracts. These works, however, focus on testing smart contracts rather than verifying them. For instance, these symbolic execution engines set a bound on the loop iterations or the number of function calls and aim to cover those bounded program paths with generated test cases. There are also several attempts on verifying smart contracts, such as Securify [27], Zeus [15], *solc-verify* [13] and VerX [25]. The first three approaches translate Solidity programs into existing intermediate languages (i.e., Datalog, LLVM and Boogie) and reuse existing verification facilities. Such approaches are based on abstract interpretation, which is known to have problems like fixed abstract domains and false alarms due to coarse over-approximation. In particular, Securify does not support numerical properties like overflow; Zeus suffers from high numbers of false alarms and *solc-verify* lacks full coverage. VerX applies delayed predicate abstraction (which is based upon symbolic execution and abstraction) to verify real-world smart contracts. However, VerX only supports external-callback-free contracts [25] and a bound on the loop iteration within a function is required.

In this work, we develop a formal verification engine called *sVerify* which is designed for Solidity programs. *sVerify* is built upon lazy annotation [21] and state-of-the-art loop invariant generation techniques [17,31]. Given a smart contract with assertions, *sVerify* automatically constructs a labeled control-flow graph (CFG) of each function. Each node in the CFG is annotated lazily with an invariant (which is initially $true$) in a property-guided (i.e. assertion-guided) way. The invariants are monotonically strengthened through sound inference rules. More importantly, invariants associated with nodes contained in explicit or implicit loops are learned automatically with a combination of concrete testing, machine learning and symbolic execution techniques, based on features specific to smart contracts. The invariants are strengthened until the assertions are verified or falsified.

sVerify has been applied to verify against the common code vulnerabilities including overflow and re-entrancy which are two important types of vulnerabilities, on two sets of 835 and 3897 smart contracts respectively. It successfully verifies or falsifies 804 contracts on the first test set in the comparison experiment with Zeus. The result shows that *sVerify* suffers fewer false alarms than Zeus. In the second test subject set, 3859 contracts are successfully evaluated by *sVerify* against *solc-verify* and VeriSol. The manual examined results on 68 contracts with more than 100 transactions regarding to overflow show that *sVerify* gets fewer false alarms than *solc-verify* and more finished contracts than VeriSol. To further evaluate *sVerify* on verifying complex smart contracts against contract-specific assertions, we systematically apply *sVerify* to 7 different kinds



(a) Contract I (buggy) (b) CFG for Contract I (c) Labeled CFG for Contract I

(d) Contract II (correct) (e) Labeled CFG for Contract II

Fig. 1. Example Contracts and corresponding labeled CFGs. (Color figure online)

of contracts that have the most balances with manually specified assertions. Three contracts have been verified successfully, and the falsified assertions reveal 2 vulnerabilities in these contracts.

To summarize, this paper makes the following contributions:

- We propose a method to verify the correctness of smart contracts through lazy annotation and invariant learning.
- We develop an end-to-end verification engine *sVerify* for Solidity contracts.
- We evaluate the effectiveness of *sVerify* with real-world smart contracts against overflow and re-entrancy vulnerabilities, and find *sVerify* can verify these contracts with fewer false alarms.

2 Overview Through Motivating Examples

In this section, we give an overview on how *sVerify* works by two example contracts (one is buggy and the other is correct, as shown in Fig. 1).

Figure 1a is a simplified version of the DAO contract. The function `withdraw` allows the investor `msg.sender` to claim back his investment and sets the investor's balance to 0. However, the `msg.sender` here is a contract account, which may be controlled by an attacker. The fallback function in this malicious contract is crafted to call back the `withdraw` function again. Note that the fallback function is invoked automatically when some Ether is transferred into the contract (triggered by line 6) according to the mechanism of Ethereum Virtual Machine (EVM). This action allows

the attacker to claim more Ether than he deserves. The assertion at line 9 which requires the balance of the contract being decreased by `amt` exactly after line 8 will be violated in such cases. This vulnerability is also referred to as re-entrancy [20]. To prevent such vulnerabilities, one of the improvement shown in Fig. 1d introduces a variable `lock` to ensure the transfer at line 9 can be executed only once. In addition, it should be noted that the variable `lock` can only be modified by the function `withdraw`. The statement at line 7 requires `lock` to be `false`, and only if this condition is satisfied, `lock` is updated to be `true` and `amt` is sent to the investor `msg.sender`. If there is a callback action again, it will be reverted by the condition at line 7, such transactions always fail. As a result, the assertion at line 10 always holds.

To verify the `toyDAO_A` contract, *sVerify* first constructs the CFG of the `withdraw` function as shown in Fig. 1b. In this CFG, nodes *root* and *stop* represent the entry and exit of the function respectively. The label on the arrow is the corresponding command in the form of line number. There are two implicit edges drawn with dashed lines in Fig. 1b. These two edges link node n_4 to node *root* and node *stop* to node n_4, which capture an inter-contract function call to the function `withdraw`. Node n_5 before the assertion statement at line 9 is an assertion node, which is labeled with the corresponding assertion $this.balance = oldq - amt$ (highlighted in red).

Based on the constructed CFG, *sVerify* infers the invariant for each node and checks whether the invariant at node n_5 implies the assertion afterwards. Figure 1c shows the invariants of node n_1–n_3 with *root* node being true. Taking node n_2 as an example, its invariant is strengthened based on the invariant associated with node n_1 and statement at l_5. That is, the new invariant is the conjunction of the original invariant (which is $true$) at n_2 and $oldq = this.balance \wedge amt = balances[msg.sender]$ (which is the constraint that must be satisfied at n_2 since n_2 can only be reached from n_1). To infer the invariant at node n_4 which is the head node of the loop starting with an implicit edge labeled with **call** $withdraw$ and ended with an edge labeled with **return**, *sVerify* invokes the *loop invariant generator* to learn an invariant. It first generates random valuations of all relevant variables (including `amt`, `oldq`, and `this.balance`), then categorizes the valuations. After that it calls the learner to generate a candidate invariant which is validated by the validator thereafter. If the candidate invariant is not valid, a counterexample in the form of variable valuations is generated and used to learn a new candidate invariant. In this example, during the invariant learning process, an error sample (`amt=1, oldq=257, this.balance=256`) is generated. With this sample, the `msg.sender` will receive 1 wei (the smallest denomination of Ether) at line 6, and possibly will call back to this function again to get another 1 wei. While the second call satisfies the assertion at line 9 (`amt=1, oldq=256, balance=255`), the first call which completes subsequently violates the assertion (`amt=1, oldq=257, balance=255`). Thus, the verification terminates and the contract is falsified.

For the fixed contract `toyDAO_B` in Fig. 1d, the corresponding labeled CFG is shown in Fig. 1e where node n_5 is the head node of the loop. Similarly, *sVerify* infers the invariant for each node and invokes the *loop invariant generator* to generate the invariant for node n_5. The *loop invariant generator* generates a valid candidate invariant $locked = true \wedge this.balance = oldq - amt$ at node n_5 after a few iterations.

Afterwards, the contract is verified since the invariant at n_5 implies the assertion $this.balance = oldq - amt$ at that node successfully.

3 Our Approach

In this section, we present our approach step-by-step in detail.

3.1 Formalization of Smart Contracts

Unlike traditional programs in which the $main()$ function is the single entry, smart contracts can be accessed from any public function once they are deployed. Thus, it is important that each function is verified separately. Without loss of generality, we define the following commands which capture a core set of sequences of EVM instructions. Readers can refer to Ethereum yellow paper [29] and KSolidity [19] etc. for further details.

Definition 1 (Command). *A command in smart contracts is defined as follows.*

$$
\begin{array}{lll}
Com & ::= & sstore(p, v) \mid sload(p) \mid x := expr \mid \textbf{if } b \mid \textbf{assert } b \mid \textbf{call } f \mid \textbf{return} \\
expr & ::= & x \mid v \mid op(expr, expr) \\
op & ::= & add \mid mul \mid sub \mid div \mid mod \\
b & ::= & true \mid false \mid iszero(expr) \mid cmp(expr, expr) \mid not\ b \mid b\ and\ b \mid b\ or\ b \\
cmp & ::= & lt \mid gt \mid eq
\end{array}
$$

$sstore(p, v)$ writes a position p with value v (i.e., a 256-bit bitvector) to storage, while $sload(p)$ reads a value of p from storage. $x := expr$ assigns the valuation of expression $expr$ to variable x. The expression $expr$ can be a variable, a value, or an arithmetic operation on two expressions such as addition add, multiplication mul, and so on. Branching command **if** b evaluates a boolean expression b which can be boolean constants $true$ or $false$. The expression also includes comparison operators like $iszero$ and cmp (lt, gt, eq) together with boolean operators (not, and, or). Assertion **assert** b asserts the boolean expression b shall be true. Commands **call** f and **return** represent a call to function f and a return to the caller respectively.

Definition 2 (Function). *A smart contract function F is a tuple $(N, root, E, \mathcal{I}, \mathcal{A})$, where N is a set of nodes (representing control locations); $root \in N$ is the entry node; $E \subseteq N \times Com \times N$ is a set of edges labeled with a command defined in Definition 1; $\mathcal{I} : N -> Pred$ is a function that labels each node N with an invariant predicate; and $\mathcal{A} : N -> Pred$ is a function which labels each node N with an assertion predicate.*

The above defines a function of a smart contract to be a labeled control-flow graph (CFG) to simplify the discussion. In practice, given a function of a smart contract C, we first compile the source code into EVM bytecode [29] and subsequently disassemble the bytecode into EVM instructions. The CFG is then constructed through simulating the stack with the instructions, i.e., to figure out the targets of all `jump` instructions. To capture control flow due to the inter-contract function calls, two implicit edges are generated by linking the call node to the root node and linking the stop node to the call

$$\text{Sstore}\ \frac{n \xrightarrow{sstore(p,v)}_e n', V' = V[storage[p] \mapsto v]}{(n,\Gamma,V) \xrightarrow{sstore(p,v)}_s (n',\Gamma,V')} \qquad \text{Assign}\ \frac{n \xrightarrow{x:=expr}_e n', V' = V[x \mapsto eval(expr,V)]}{(n,\Gamma,V) \xrightarrow{x:=expr}_s (n',\Gamma,V')}$$

$$\text{If-T}\ \frac{n \xrightarrow{\textbf{if } b}_e n'}{(n,\Gamma,V) \xrightarrow{\textbf{if } b}_s (n',\Gamma,V)} \qquad \text{If-F}\ \frac{n \xrightarrow{\textbf{if } !b}_e n''}{(n,\Gamma,V) \xrightarrow{\textbf{if } !b}_s (n'',\Gamma,V)}$$

$$\text{Call}\ \frac{n \xrightarrow{\textbf{call } f}_e n', V' = extract(V)}{(n,\Gamma,V) \xrightarrow{\textbf{call } f}_s (n',\Gamma^\frown\langle(f,V_\Gamma)\rangle,V')} \qquad \text{Return}\ \frac{n \xrightarrow{\textbf{return}}_e n', V' = V \oplus V_\Gamma}{(n,\Gamma^\frown\langle(f,V_\Gamma)\rangle,V) \xrightarrow{\textbf{return}}_s (n',\Gamma,V')}$$

Fig. 2. Execution rules, where $(n \xrightarrow{c}_e n') \in E$

node. Through these, a complete CFG is constructed. Readers are referred to [3,6] for further details.

Initially, the invariant function $\mathcal{I}$ is defined such that $\mathcal{I}(n) = true$ for every $n \in N$. Furthermore, the assertion function $\mathcal{A}$ is defined such that $\mathcal{A}(n) = b$ if n is a program location with a command **assert** b; otherwise $\mathcal{A}(n) = true$. For instance, as shown in the CFG of function $withdraw$ in Fig. 1e, the invariant $\mathcal{I}(n_5)$ of node n_5 is $true$, and the assertion $\mathcal{A}(n_5)$ is $this.balance = oldq - amt$.

Definition 3 (Symbolic Semantics). *Let $(N, root, E, \mathcal{I}, \mathcal{A})$ be a function of a smart contract, its (symbolic) semantics is defined as a labeled transition system $(S, init, \rightarrow_s, \mathcal{I}, \mathcal{A})$, where S is a set of symbolic states, and each state s is a triple (n, Γ, V) where $n \in N$, Γ is a call stack,[1] and V is a symbolic valuation function which maps program variables to expressions of symbolic variables, $init \in S$ is the initial state, $\rightarrow_s \subseteq S \times Com \times S$ is the transition relation of the semantics while E is the transition relation at the code level. $\rightarrow_s$ conforms to the semantic rules defined in Fig. 2.*[2]

In Fig. 2, rule $Sstore$ captures how the value of the position in storage is updated. After the execution of the command, n is moved to the next node n' and position p in storage V' is updated by the value of v. Rule $Assign$ updates the value of variable x in V' based on the evaluation of expression $expr$ in the valuation V (denoted by function $eval$). The rules of *If-T* and *If-F* capture the branch situation, n is moved to either node n' or n'' after executing this command. Rule $Call$ captures the execution of any possible inter-contract function call. After the execution, n is moved to the root node of the called function n', function f and the valuation of the local variables V_Γ are added to the function call stack $\Gamma^\frown\langle(f, V_\Gamma)\rangle$, and valuation V' is to extract the valuation of global variables in V that are only modified in current function by $extract$. Rule $Return$ pops the top element of the stack and moves to the node of the caller with the updated valuation which restores the local variable valuation at the calling node. Symbol $\oplus$ overrides the variable valuation in V with those in V_Γ.

[1] We omit the details on the content of the stack for brevity.

[2] Due to the page limit, only a core set of rules are presented here.

A path p of a function in a smart contract is a sequence of alternating nodes/commands in the form of $\langle n_0, c_0, n_1, c_1, \ldots, c_n, n_{n+1}\rangle$, where $n_0 = root$ and $n_i \xrightarrow{c_i}_e n_{i+1}$ for all $0 \leq i \leq n$. A (symbolic) trace is a path in the symbolic semantics, and each trace corresponds to a path in the contract by definition. Thus, a trace tr is a sequence of alternating states/commands in the form of $tr = \langle s_0, c_0, s_1, c_1, \ldots, c_n, s_{n+1}\rangle$, where $s_0 = init$ and $s_i \xrightarrow{c_i}_s s_{i+1}$ for all $0 \leq i \leq n$. We write $last(tr)$ to denote the last state of the trace s_{n+1}. The set of symbolic traces of a function F, written as $Trace(F)$, is the set of traces of its symbolic semantics, where each trace is a sequence whose head is the initial state and the alternating state/command conforms to the transition relation.

Definition 4 (Node Invariant). *Given a smart contract function $F = (N, root, E, \mathcal{I}, \mathcal{A})$, a predicate ϕ is an invariant at node n (denoted as $\mathcal{I}(n) = \phi$) if and only if $last(tr) \models \phi$ for all $tr \in Trace(F)$ s.t. $\pi(last(tr)) = n$.*

where $s \models \phi$ means ϕ is satisfied by the variable valuation s. Intuitively, the above definition states ϕ is an invariant at node n if and only if ϕ is satisfied by all traces leading to node n, i.e., when the trace reaches n, its variable valuation satisfies ϕ. Function $last$ returns the last element of the trace, and function π returns the node of the tuple.

Definition 5 (Contract Correctness). *Given a contract C with each function $F_i = (N_i, root_i, E_i, \mathcal{I}_i, \mathcal{A}_i)$, F_i is correct if $\forall n_j \in N_i,\ \mathcal{I}_i(n_j) \Rightarrow \mathcal{A}_i(n_j)$. Contract C is correct if all the functions F_i in C are correct.*

Based on the constructed CFG and its semantics, the verification of a smart contract can be achieved by checking whether the invariant of any node can imply the associated assertion. If yes, the program is verified to be correct. *sVerify* infers the node invariants with the method of strongest postcondition. Before presenting how the inference works, we first define how the strongest postcondition is computed.

Definition 6 (Strongest Postcondition). *Given a command $c \in Com$ and a precondition ϕ, the strongest postcondition $sp(c, \phi)$ is defined as:*

$$
\begin{aligned}
&sp(sstore(p, v), \phi) = \exists y,\ \phi[y/storage[p]] \wedge storage[p] = v \\
&sp(x := expr, \phi) = \exists y,\ x = expr[y/x]\ \wedge\ \phi[y/x] \\
&sp(c, \phi) = \phi \wedge b \qquad\qquad \text{if } c = \textbf{if } b \text{ or } \textbf{assert } b \\
&sp(c, \phi) = \phi \qquad\qquad \text{if } c = sload(x) \\
&sp(\textbf{call } f, \phi) = \forall x \in LV,\ \forall y \in GV',\ \phi \ominus \phi(x) \ominus \phi(y)
\end{aligned}
$$

In the above definition, the fresh variable y represents the previous values of $storage[p]$ and x in the strongest postconditions for command $sstore$ and assignment. For the branching and assertion commands, the strongest postcondition is the conjunction of ϕ and b. As command $sload$ only reads the storage, its strongest postcondition keeps the same. We remark that the strongest postcondition for command $\textbf{call } f$ is ϕ except that all constraints related to local variables LV and global variables GV' are eliminated. Symbol $\ominus$ represents variable elimination of all variables in ϕ. GV' is global variables

Algorithm 1: Node Invariant Inference Algorithm $inferI(F, n)$

```
1 Ψ ← false;
2 for (m_i, c_i, n) ∈ E do
3 |   Ψ ← Ψ ∨ sp(c_i, I(m_i)) ;
4 end
5 if Ψ ≠ false then I(n) ← I(n) ∧ Ψ ;
```

which can be modified by other functions besides the current function. This rule can be potentially improved with a contract-level invariant inference method. In *sVerify*, we conduct basic static analysis which allows us to identify the global variables that are modified by each function in the contract. With that information, we strengthen the above rule as follows: all constraints on global variables except those which are only modified by the current function, are eliminated. This is sound as all callback actions to the current function are captured in the CFG.

Algorithm 1 shows details on updating the invariant of a node n based on the strongest postcondition. Let Ψ be a predicate which is initially $false$. We compute $sp(c_i, \mathcal{I}(m_i))$ for each transition (m_i, c_i, n) to node n by command c_i. Their disjunction is a constraint which must be satisfied by the invariant at node n. Intuitively, this is because n can only be reached via one of its parents. Lastly, at line 5, we set the invariant at node n to be the conjunction of $\mathcal{I}(n)$ and Ψ so that it is monotonically strengthened over time. The condition at line 5 ensures that a node without a parent like the root node is not updated.

Proposition 1. *The invariant inferred by Algorithm 1 is indeed an invariant.* □

3.2 Loop Invariant Generation

While Algorithm 1 can be applied to infer invariants systematically, it may not be effective for loops. That is, given a loop of the form $\langle n_0, c_0, n_1, c_1, n_2 \ldots, n_k, c_k, n_0 \rangle$, the invariant of node n_0 is recursively inferred based on itself and thus may never terminate. Therefore, we distinguish head nodes of certain loops (i.e., a node representing the start of a loop statement or an external function call, it can be identified from the CFG) and apply a different approach to infer invariants for such nodes. The overall idea is an iterative "guess and check" approach for synthesizing loop invariants. This iterative approach consists of three phases, *data labeling*, *learning* (or guessing), and *validation*. The details are shown in Algorithm 2 where F is the CFG of the function and n is the head node of a loop.

In Algorithm 2, Var is the set of loop-related variables. The valuation set of variables in Var at node n (denoted as DS) is initiated by random sampling at line 1 and the size of the initial DS is decided empirically, e.g., 20. Note that an effective sampling method would allow us to learn the invariant efficiently, as shown in [17]. On the other hand, since the learned invariant is always validated by the validator, the learning is guaranteed to converge if there exists an invariant of the supported form. In general, a reasonably large set of random samples is often helpful in learning candidate invariants.

Algorithm 2: $genLI(F, n)$

```
1  DS = init(Var); DS' ← ∅; LDS ← ∅;
2  for ds ∈ DS do
3  |   DS' ← DS' ∪ concExLP(ds, n);
4  end
5  LDS ← label(DS', F, N) ;
6  while not timeout do
7  |   (flag, ds) ← checkErr(LDS);
8  |   if !flag then
9  |   |   return ("falsified", ds);
10 |   end
11 |   φ ← learnINV(LDS);
12 |   CE ← validate(φ, F, n);
13 |   if CE = ∅ then
14 |   |   return ("succeed", φ);
15 |   else
16 |   |   for ds ∈ CE do
17 |   |   |   DS' ← DS' ∪
   |   |   |     concExLP(ds, n);
18 |   |   end
19 |   |   LDS ← label(DS', F, N) ;
20 |   end
21 end
22 (CE, φ') ← heurAndVal(φ, F, n);
23 if CE = ∅ then return ("succeed", φ') ;
24 else return ("timeout", null) ;
```

Algorithm 3: Overall Algorithm

```
1  {F_1, F_2, ..., F_m} ← CFG_build(C);
2  for F ∈ {F_1, F_2, ..., F_m} do
3  |   I' ← ∅; I ← {true | n ∈ N};
4  |   while I' ≠ I do
5  |   |   I' ← I;
6  |   |   for n ∈ N do
7  |   |   |   if n is loop head then
8  |   |   |   |   (msg, v) ← genLI(F, n);
9  |   |   |   |   if msg = "succeed" then
10 |   |   |   |   |   I(n) ← v;
11 |   |   |   |   else if msg = "falsified" then
12 |   |   |   |   |   return ("falsified", v);
13 |   |   |   |   else
14 |   |   |   |   |   return ("timeout", null);
15 |   |   |   else
16 |   |   |   |   I(n) ← inferI(F, n) ;
17 |   |   |   end
18 |   |   end
19 |   end
20 |   for n ∈ N do
21 |   |   if I(n) ⇏ A(n) then
22 |   |   |   return ("falsified", ce)
23 |   |   end
24 |   end
25 end
26 return "verified" ;
```

LDS is labeled DS', which is updated at line 5 by $label$ function. The data samples are collected through lines 2–4 by concretely executing the loop part with the valuations from DS. During the execution, node n may be visited iteratively and all the variable valuations upon reaching n are added to DS' as well. Labeling for valuations in DS' is based on three categories, i.e., '+' for positive, '$-$' for negative, and 'e' for error. A valuation s which starts from an initial valuation s_0 and becomes s after zero or more iterations is labeled based on whether s_0 satisfies $\mathcal{I}(n)$ and whether eventually an assertion is violated. Specifically,

- '+': if s_0 satisfies $\mathcal{I}(n)$, and no assertion is violated during the execution.
- '$-$': if s_0 violates $\mathcal{I}(n)$ and an assertion is violated during the execution.
- 'e': if s_0 satisfies $\mathcal{I}(n)$, and an assertion is violated during the execution.

Intuitively, the valuations labeled with '+' must satisfy the (unknown) loop invariant; the one labeled with '$-$' must not satisfy the loop invariant; and a valuation labeled with 'e' is a concrete counterexample which falsifies the assertion. Take the contract in Fig. 1d as an example, assume 2 valuations $(2, 0, 20, 18)$, $(5, 1, 30, 25)$ for variables (`amt`, `lock`, `oldq` and `this.balance`) are randomly sampled at line 1. After executing function $concExLP$ with these valuations at line 3, 1 more valuation is added to DS': $(2, 0, 20, 16)$, which violates the assertion. Afterwards, valuation $\{(5, 1, 30, 25)\}$ is labeled with '+'; and $\{(2, 0, 20, 18), (2, 0, 20, 16)\}$ are labeled with '$-$'.

After labeling the initial dataset, we try to strengthen a valid invariant from lines 6–21. Lines 7–10 check whether there is any 'e' valuation and return "falsified" together with the valuation as a counterexample. 'e' valuation is a concrete valuation in LDS which violates any assertions. A candidate invariant is expected from function

$learnINV$ at line 11. The primary idea is to guess a candidate invariant in the form of a classifier which separates the valuations labeled with '+' from those labeled with '−'. Specifically, we adopt the LINEARARBITRARY algorithm proposed in [31], which is built upon SVM and decision tree classification, to infer candidate invariants in the form of arbitrary combination of conjunction or disjunction of linear inequalities. Line 12 invokes the function $validate$ to check whether the candidate invariant ϕ is indeed an invariant (i.e., it is inductive through every path in the loop). That is, we tentatively label the node n with the candidate and apply Algorithm 1 to propagate it through nodes starting from n and ending with a parent of n. The invariant is inductive if and only if, for all m such that $(m, c, n) \in E$, $sp(\mathcal{I}(m), c) \Rightarrow \phi$, which means ϕ is a valid invariant and returned at line 14. Otherwise, a counterexample in the form of variable valuation is generated and added to CE, which is further subsumed into LDS for the next round invariant generation.

We remark that the loop invariants learned through this way are property-guided. Although the learning algorithm adopted from [31] is guaranteed to terminate given a finite set LDS, the overall learning process may timeout due to too many guess-and-check iterations. We adopt a simple heuristics of conjuncting the assertion with the current candidate as a candidate invariant for validation at line 22. This is justified intuitively as the learned invariant should be strong enough to imply the assertion. For example, in contract `toyDAO_B` shown in Fig. 1d, a candidate invariant $lock = true$ is generated by Algorithm 2. However, timeout occurs when *sVerify* validates it. Applying the heuristics, the candidate invariant is strengthened to be $lock = true \wedge this.balance = oldq - amt$, which is subsequently validated. Otherwise, timeout is returned at line 24.

3.3 Overall Verification Algorithm

With the above discussion, we are ready to present the overall algorithm which is shown in Algorithm 3. Given a smart contract C with m functions, we first construct the CFG for each function at line 1. For each node n in each function F, we initiate the node's invariant with $true$ and update them at lines 4–19. If node n is a loop head node, Algorithm 2 is invoked and an invariant is returned when it is "succeed" at line 10. Otherwise, the algorithm will return "falsified" or "timeout" at lines 12 and 14. Whenever the invariants stabilize (i.e., reaches a fixed point), we check whether, for each node, its invariant implies its assertion at lines 20–24. If the implication fails at any node, the counterexample (ce) from the SMT solver that violates the node assertion is returned to the user. If all assertions are implied by their corresponding invariants, the contract is successfully verified.

Theorem 1. *The contract is safe if Algorithm 3 returns "verified".*

Proof. The claim follows the fact that all inferred invariants are indeed invariants. There are two ways of inferring invariants, either by Algorithm 1 or 2. In the former case, the inferred invariant is indeed an invariant according to Proposition 1. In the latter case, the correctness of the inferred invariant generated by $genLI$ is ensured by function $validate$ in Algorithm 2 which checks whether the learned invariant is inductive. Given that all inferred invariants are sound, Algorithm 3 is sound as it returns "verified" when all assertions are implied by the invariants (by Definition 5). □

In practice, Algorithm 3 is made always terminating with a timeout on the $genLI$ method. The complexity of the algorithm is hard to analyze due to the many components. We thus evaluate it empirically in the next section.

4 Implementation and Evaluations

We have implemented our approach in *sVerify* with C++. Given a Solidity smart contract, *sVerify* first compiles it into EVM bytecode and subsequently disassembles the bytecode into instructions for constructing the CFG. Then, LIBSVM [5] and C5.0 [26] are adopted for invariant learning, and Z3 SMT solver is used for invariant validation. We conduct two sets of experiments to evaluate *sVerify* on real-world smart contracts. In particular, we attempt to address the following two questions.

1. How effective is *sVerify* in verifying common code vulnerabilities?
2. How effective is *sVerify* in verifying contract-specific assertions?

All experiments are conducted on a machine with an Intel Core i7-7700HQ CPU with 8 cores clocked at 2.8 GHz, and 23.4 GB of RAM, running the system of 64-bit Ubuntu 18.04LTS. The dependancies of *sVerify* include Z3 (version 4.8.0) and the boost library (version 1.68.0). As of now, it is developed for Solidity before version 0.5.19 and Ethereum Virtual Machine (EVM) before version 1.8.21.

4.1 Verification Against Common Code Vulnerabilities

In this set of experiments, we evaluate the performance of *sVerify* on verifying against common code vulnerabilities including overflow and re-entrancy. These two kinds of vulnerabilities are particularly interesting and relevant.

First, most of the vulnerabilities (90.2% $(476/528)$) reported in the CVE list [9] between 2018 and 2020 are overflow problems. The DAO attack [1], one of the most famous attacks which caused huge monetary loss, has evidenced the importance of re-entrancy. Furthermore, re-entrancy is a vulnerability which is associated with implicit loops due to fallback function calls and thus would put our loop invariant generation approach under test. Assertions for capturing overflow vulnerabilities are systematically generated and assertions for capturing re-entrancy vulnerabilities are manually specified regarding the balance after each call transaction like the example in Sect. 2.

For baseline comparison, we focus on three state-of-the-art verification tools Zeus, *solc-verify* and VeriSol. Zeus [15] is a framework for automatic verification of smart contracts based on abstract interpretation techniques. *solc-verify* [13] and VeriSol [28] are tools that allow specification and modular verification of Solidity contracts which are built upon the Boogie verifier.

Setup. To compare with Zeus, we adopt the test subjects reportedly analyzed by Zeus in [15] and systematically run *sVerify* on them. We did not compare with the other two tools because (1) solc-verify lacks the support of complex data types and memory models before version 0.5.0 and thus fails to verify most of the test subjects; (2) VeriSol

Table 1. Comparison results between Zeus and *sVerify*

	Zeus					*sVerify*				
Category	Safe	Unsafe	Unk.	FP	FN	Safe	Unsafe	Unk.	FP	FN
Overflow	234	592	9	33	5	255	549	31	4	0
Re-entrancy	803	28	4	2	20	754	50	31	0	0

```
function split() payable public {
   uint fee = msg.value / 100; ...
   etcDestination.call.value(msg.value - fee)(); }
function process(bytes32 _destination) payable returns (bool) {
  if (msg.value < 100) throw;
  var tax = msg.value * taxPerc / 100; ... }
function testNumberRequest(address randomreality, ...) payable {...
   uint256 cost = randomrealityapi.getPrice(200000);
   bytes32 id = randomrealityapi.requestNumber.value(cost)(...);  ... }
function transferFrom() returns (bool success) {
   ... if(now < startTime + 1 years) ... }
function multisend(..., address[] dests, uint256[] values){ ...
   while (i < dests.length) {
      assert( i < values.length);
      ERC20(_tokenAddr).transfer(dests[i], values[i]);
      i+=1; ... }
```

Fig. 3. Functions incorrectly analyzed by tools

requires manual-specified assertions for specific properties and thus we leave the comparison to the second experiment. Note that the code of Zeus is not open source and thus it is not possible to apply it to other smart contracts. Among 1524 contracts reportedly analyzed by Zeus, 898 of them are still available online. As nested loops are yet to be supported mainly due to the required engineering effort as well as lack of motivation - there are relatively small amount of nested loop contracts on the blockchain. Thus, the remaining 835 contracts are taken as the test subjects.

We further evaluate *sVerify* on 3897 contracts against open-source tools, *solc-verify* with version of v0.4.25-boogie to include the support of arithmetic mod-overflow and VeriSol[3] of 0.1.5-alpha. Note that only 68 contracts that have more than 100 transactions are demonstrated in the paper, which are also the same subjects discussed by *solc-verify* [13]. The option of flag "arithmetic" for *solc-verify* is "mod-overflow". Similar flag with the option of "useModularArithmetic" is also set for VeriSol. Timeout for verifying each contract is 3600 s for all tools. Furthermore, a 10 s timeout is set for each z3 solver request.

Results. The experiment results on Zeus's 835 test subjects are summarized in Table 1.[4] Each result is either "Safe" or "Unsafe" (i.e., there is a potential issue). "Unk." means unknown, due to either exception or timeout. "FP" and "FN" stand for false positives and false negatives. A false positive occurs when tools return "Unsafe" but the contract

[3] Necessary assertions regarding overflow and reentrancy are inserted manually.
[4] Details and benchmarks can be found at https://doi.org/10.5281/zenodo.5168441.

Table 2. Comparison results on Overflow with *solc-verify* and VeriSol.

	solc-verify					VeriSol					*sVerify*				
Category	Safe	Unsafe	Unk.	FP	FN	Safe	Unsafe	Unk.	FP	FN	Safe	Unsafe	Unk.	FP	FN
Overflow	34	34	0	30	0	18	12	38	1	0	41	25	2	9	0

is actually "Safe" after we manually examined the alarmed code, while a false negative occurs when tools return "Safe" but the contract is actually "Unsafe".

We have multiple observations based on the results. First, compared with Zeus, *sVerify*'s verification results are more reliable since there are fewer false positives and false negative. In particular, for overflow, Zeus generates 33 false positives and 5 false negatives, whereas *sVerify* has 4 false positives and 0 false negative; for re-entrancy, *sVerify* has 0 false positive and 0 false negative.

Since Zeus is not open source, there is no way to know why some contracts are not correctly analyzed. We show some examples in Fig. 3 in the following which may offer clues. Zeus generates a false alarm of overflow for function `split` in Fig. 3 which sends tokens to two accounts. We speculate the false alarm is due to line 3, since examining this line alone would suggest that overflow is possible due to the arithmetic operation. In comparison, *sVerify* keeps track of relationship between `fee` and `msg.value` due to line 2 and correctly concludes there is no overflow. Zeus misses the overflow in function `process` where the statement `msg.value*taxPerc/100` may exceed the maximum value at line 6. For re-entrancy, one example Zeus misses is the one in function `testNumberRequest` where attackers may input some address to exploit the re-entrancy vulnerability at line 9. The reason of four false positives by *sVerify* is because *sVerify* verifies each function in isolation. Namely, symbolic values are assigned to global variables so that they may have arbitrary values. In reality, these variables may be constrained in certain ways. For instance, `startTime` in function `transferFrom` is only set in constructor and the overflow at line 11 is impossible. Finally, we notice that *sVerify* missed reporting one re-entrancy vulnerability as *sVerify* terminates the analysis once an issue is identified, e.g., an overflow issue is identified before a re-entrancy issue is encountered.

Table 2 demonstrates the results of 68 contracts by three tools. It can be observed that *solc-verify* has more false positives compared to VeriSol and *sVerify*. There are multiple reasons why false alarms are generated by *solc-verify*. For instance, missing range assumptions for array lengths causes false alarms for loop counters [13], which contributes the most false alarms. On the contrary, *sVerify* identifies more true vulnerabilities. One example is the function `multisend` shown in Fig. 3. *solc-verify* reports `i+=1` might overflow, which is regarded as a false alarm (FP). However, *sVerify* reports the index of variable `values` at line 15 might cause overflow if `i` is larger than the length of array `values`, which is a true issue that is missed by *solc-verify*. VeriSol can also find such problems if only an assertion shown at line 14 is inserted. Only 30 contracts are successfully analyzed by VeriSol. 9 false alarms are generated by *sVerify*. Besides the missing constraints on time like the case in func-

Table 3. Real-world Contracts Analysis.

Contract	#loc	#pubfns	#lpfns	*sVerify*	*solc-verify*
MultisigWallet	304	14	7	Unsafe	Unk.
Imt	65	4	1	Safe	FP
WithdrawDAO	15	2	0	FP	FP
LifCrowdsale	800	37	1	Safe	Unk.
WETH	50	6	0	FP	FP
KyberReserve	298	19	2	Safe	Unk.
TokenStore	240	20	3	Unsafe	Unk.

tion `transferFrom`, other run-time variables also matter like the amount of Ether in `amt=msg.value*2000`, which is safe as the total Ether is limited.
Efficiency sVerify successfully analyzed 804 (out of 835) contracts and 3859 (out of 3897) contracts for two sets of benchmarks, and each contract takes an average of 38.5 s and 14.8 s respectively. On the contrary, Zeus finishes 97% of the contracts within 60s, there is no further detailed data provided. *solc-verify* finishes all the contracts with an average time of 1.24 s and VeriSol finishes 30 (out of 68) contracts with 2.28 s. Longer time is needed to learn invariants for loops in the verification process, which is an essential step to acquire an accurate result, but also leads to more timeouts.

4.2 Verifying Contract-Specific Assertions

While verification against common vulnerabilities is important, it is far from sufficient for the functional correctness. In this section, we identify several high-profile smart contracts, manually specify assertions relevant to their functional correctness and apply *sVerify* to verifying those assertions. The assertions are mainly targeted at functions with loops as those are non-trivial to verify. Since most of the loops operate on arrays, we define several patterns specific to them, e.g., `assert(ret==ARRAY_MAX)` to check whether the returned value `ret` by the program is the maximum. The test subjects consist of 7 representative contracts from accounts ranking top 1000 in terms of balance, including the wallet contracts which receive and transfer Ether for users, like *multiSig*, *Imt* and *WithdrawDAO*, the token contracts which work for token issuance and crowdsale, like *LifCrowdsale* and *WETH*, the decentralized exchange contracts which work for crypto asset transaction, like *KyberReserve* and *TokenStore*. Many contracts are built upon these contracts. Table 3 shows the results of *sVerify* and *solc-verify*, where columns $\#loc$, $\#pubfns$, and $\#lpfns$ stand for lines of code, number of public functions, and number of loop functions. The results by VeriSol are ignored because of version problem.

sVerify successfully analyzes all the contracts whereas *solc-verify* finishes three. Two out of four alarms reported by *sVerify* are real vulnerabilities. In function `getTxIds` shown in Fig. 4, the statement at line 2 overflows if the assigned value

```
1 function getTxIds(uint from, uint to, ...) public returns(uint[] _txIds){ ...
2     _txIds = new uint[](to - from); ... }
3 function withdraw(uint _amount){
4     ... tokens[0][msg.sender] = safeSub(tokens[0][msg.sender], amount);
5     uint oldq = this.balance;
6     if (!msg.sender.call.value(amount)()) ...
7     assert(this.balance == oldq - amount); }
```

Fig. 4. Alarmed functions by *sVerify*

of variable `to` is smaller than `from` (which may spawn new arrays and cost up all the gas). The other one is in function `withdraw`, the assertion statement at line 7 is violated if the fallback function in `msg.sender` calls back to function `withdraw` again. Although this, in practical runtime, the smart contract decreases the token amount of the `msg.sender` at line 4, which ensures the `msg.sender` cannot claim more Ether than he deserves. The other two false alarms are due to limitations on analyzing functions in isolation, as explained for the constraints of time and Ether balance in Sect. 4.1. Two false alarms are all eliminated after inserting `require` statements for restricting the arithmetic overflow. In comparison, *solc-verify* reports 3 alarms which are all false alarms. This test shows that *sVerify* can be helpful to verify some contract-specific assertions.

5 Related Work

In the last five years, several approaches have been proposed to test or verify smart contracts through various techniques. For instance, the fuzzing tools reported in [2,14,30] try to selectively generate test inputs with both static and dynamic techniques to find critical vulnerabilities. Inevitably, they are prone to false negatives which are of great concern for verification of smart contracts. Other works adopt symbolic techniques to analyze smart contracts [8,20,22,24]. To avoid the path explosion problem, these approaches usually bound the search space by, for instance, setting a limit on the number of blocks or function calls.

Unlike these approaches, Securify [27] is based on abstract interpretation and dependency graph to produce vulnerability patterns through inference rule-based generation and analyze the correctness accordingly. However, Securify does not support numerical properties like overflow. VerX [25] introduces delayed predicate abstraction approach based upon symbolic execution to verify smart contracts during transaction execution. However, VerX only supports external-call-free contracts whose behavior is equivalent to the behavior of the contracts without callbacks. Some other approaches like the work in [4] and Zeus [15] translated smart contracts into intermediate representations like F* programs and LLVM bitcode respectively, then leverage existing tools for F* and Seahorn to reason about contract correctness. *solc-verify* [13] and verisol [28] translate smart contracts into the Boogie intermediate language, and leverages the verification toolchain for Boogie programs for analysis. The translation is on the source code level, which allows the users to write annotations directly in the contract. However, since Boogie was not designed for smart contracts, some features are not supported for the translation.

In this work, we propose a verification approach based on lazy annotation and automatic loop invariant generation. A number of loop invariant generation approaches have been proposed, including those based on abstraction interpretation [11], counterexample-guided abstraction refinement [7] or interpolation [18], logical inference [10] and learning [17,31]. The former three depend on constraint solving and thus suffer from scalability. We adopt the learning-based invariant generation approach in this work.

6 Conclusion

We leverage the techniques of lazy annotation and state-of-the-art loop invariant generation method to implement the formal verifier *sVerify*. With the help of invariant inference, *sVerify* can be helpful to verify or falsify smart contracts. We evaluated *sVerify* on 4670 real-world smart contracts and the results show that *sVerify* is effective and reasonably efficient. We will extend our work to contract-level verification in the future.

References

1. Dao (2016). https://www.coindesk.com/understanding-dao-hack-journalists
2. Akca, S., Rajan, A., Peng, C.: SolAnalyser: a framework for analysing and testing smart contracts, pp. 482–489 (2019). https://doi.org/10.1109/APSEC48747.2019.00071
3. Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A.: Analyzing smart contracts: from EVM to a sound control-flow graph. arXiv preprint arXiv:2004.14437 (2020)
4. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N.: Formal verification of smart contracts: short paper. In: PLAS, pp. 91–96. ACM (2016)
5. Chang, C.C., Lin, C.J.: LIBSVM: a library for support vector machines. ACM TIST **2**, 27:1–27:27 (2011). http://www.csie.ntu.edu.tw/~cjlin/libsvm
6. Chang, J., Gao, B., Xiao, H., Sun, J., Cai, Y., Yang, Z.: sCompile: critical path identification and analysis for smart contracts. In: Ait-Ameur, Y., Qin, S. (eds.) ICFEM 2019. LNCS, vol. 11852, pp. 286–304. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32409-4_18
7. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003)
8. ConsenSys: Mythril: Security analysis of ethereum smart contracts (2018). https://github.com/ConsenSys/mythril. Accessed 30 May 2019. online
9. CVE: CVE list. https://cve.mitre.org/data/downloads/index.html. Accessed 4 June 2021
10. Dillig, I., Dillig, T., Li, B., McMillan, K.: Inductive invariant generation via abductive inference. In: OOPSLA, pp. 443–456 (2013)
11. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL, pp. 191–202. ACM (2002)
12. Haber, S., Stornetta, W.S.: How to time-stamp a digital document. In: Menezes, A.J., Vanstone, S.A. (eds.) CRYPTO 1990. LNCS, vol. 537, pp. 437–455. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-38424-3_32
13. Hajdu, Á., Jovanović, D.: SOLC-VERIFY: a modular verifier for solidity smart contracts. In: Chakraborty, S., Navas, J.A. (eds.) VSTTE 2019. LNCS, vol. 12031, pp. 161–179. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-41600-3_11
14. Jiang, B., Liu, Y., Chan, W.: ContractFuzzer: fuzzing smart contracts for vulnerability detection, pp. 259–269 (2018). https://doi.org/10.1145/3238147.3238177

15. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: NDSS. The Internet Society (2018)
16. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
17. Li, J., Sun, J., Li, L., Le, Q.L., Lin, S.: Automatic loop-invariant generation and refinement through selective sampling. In: ASE, pp. 782–792 (2017)
18. Lin, S., Sun, J., Nguyen, T.K., Liu, Y., Dong, J.S.: Interpolation guided compositional verification (t). In: ASE, pp. 65–74 (2015)
19. Lin, S.: K-framework Solidity (2018). https://github.com/kframework/solidity-semantics
20. Luu, L., Chu, D.H., Olickel, H., Saxena, P.: Making smart contracts smarter. In: CCS, pp. 254–269. ACM (2016)
21. McMillan, K.L.: Lazy annotation for program testing and verification. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 104–118. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_10
22. Mossberg, M., Manzano, F., Hennenfent, E., Groce, A.: Manticore: a user-friendly symbolic execution framework for binaries and smart contracts. In: ASE, pp. 1186–1189. IEEE (2019)
23. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system. Technical report, Manubot (2019)
24. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: ACSAC, pp. 653–663. ACM (2018)
25. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: VerX: safety verification of smart contract. In: IEEE Symposium on Security and Privacy (2020)
26. Quinlan, J.: C5.0: an informal tutorial (2017). http://www.rulequest.com/see5-unix.html
27. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M.: Securify: practical security analysis of smart contracts. In: CCS, pp. 67–82. ACM (2018)
28. Wang, Y., et al.: Formal verification of workflow policies for smart contracts in azure blockchain. In: Chakraborty, S., Navas, J.A. (eds.) VSTTE 2019. LNCS, vol. 12031, pp. 87–106. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-41600-3_7
29. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper **151**, 1–32 (2014)
30. Wüstholz, V., Christakis, M.: Harvey: a greybox fuzzer for smart contracts. In: ESEC/FSE, pp. 1398–1409 (2020)
31. Zhu, H., Magill, S., Jagannathan, S.: A data-driven CHC solver. In: PLDI, pp. 707–721. ACM (2018)