sVerify: Verifying Smart Contracts through Lazy Annotation and Learning

Anonymous Author(s)

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 from 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 novel 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 both effective and efficient.

ACM Reference Format:

Anonymous Author(s). 2020. sVerify: Verifying Smart Contracts through Lazy Annotation and Learning. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Blockchain [7, 35] is a fast-growing research area in recent years. It is first conceptualized in Bitcoin blockchain [34] by Satoshi Nakamoto based on multiple techniques like cryptographic chain of blocks by Stuart Haber and W. Scott Stornetta [19], distributed systems by Lamport [25] etc. The emergence of Bitcoin makes financial transactions among strangers possible without the help of third-party authority. Later on, Buterin stepped forward to develop the platform Ethereum [45], 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 [42]. It can be regarded as a computer program, which is mostly 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 [48]. 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 equivalent to 60 million USD at the time. The attacker found a loophole in the *splitDAO* function so that he could repeatedly withdraw Ether over and over again through an implicit loop in the *fallback* function in a single transaction.

With the increasing amount of attacks on smart contracts, various approaches and tools have been developed to analyze the correctness of smart contracts. For instance, Luu *et. al* [29] developed a symbolic execution engine for Solidity smart contracts called Oyente, which systematically analyzes individual functions in a smart contract to identify vulnerabilities. Nikolic *et al.* [36] 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. The above-mentioned 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. Existing approaches which may be applied to verify smart contracts include Securify [43], Zeus [24], SOLC-VERIFY [20] and VerX [39]. The first three approaches translate Solidity programs into existing intermediate languages (i.e., Datalog, LLVM and Boogie) and reuse existing verification facilities. Such approaches have two limitations. First, since the verification facilities are not designed for smart contracts, they can only play a limited role on some specific properties of smart contracts. Second, these 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 cannot verify 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 during transaction execution. However, VerX only supports external-callback-free contracts 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 [32] and state-of-the-art loop invariant generation techniques [27, 47]. 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*). The invariants are monotonically strengthened through sound inference rules. More importantly, invariants associated with nodes contained in explicit or implicit

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 common code vulnerabilities, 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 against overflow and re-entrancy in the comparison experiment with Zeus and SOLC-VERIFY. The result shows that *sVerify* suffers from fewer false alarms than Zeus. In particular, *sVerify*'s verification results on 99.9% of the smart contracts are correct, whereas Zeus's are correct on 96.3% of the smart contracts. Compared with SOLC-VERIFY, *sVerify* is much more scalable in this set, i.e., *sVerify* verifies or falsifies 804 contracts whereas SOLC-VERIFY finishes only 56 of them. In the second test subject set, 3859 contracts are successfully evaluated by *sVerify*. The results show that *sVerify* gets a better correct rate of 88.2% than that of 57.4% by SOLC-VERIFY on the 68 contracts with more than 100 transactions regarding to overflow.

To further evaluate *sVerify* on verifying complex smart contracts against contract-specific assertions, we systematically apply *sVerify* to 7 different kinds of contracts that have the most balances with manually specified assertions. All except 3 assertions have been verified successfully after inserting requirements for capturing the overflow alarms, and the falsified assertions lead to the discovery of 2 vulnerabilities in the contracts.

Our contributions are summarized in the following.

- We develop *sVerify*, which is an end-to-end verification engine directly supporting Solidity.
- We propose a new way to learn loop invariants which support implicit loops due to fallback function calls in smart contracts.
- We evaluate the effectiveness of *sVerify* with 4670 real-world contracts.

The remainders of the paper are organized as follows. In Section 2, we present an overview of our approach and illustrate how *sVerify* works through an example. In Section 3 we present the details of *sVerify* step-by-step. Section 4 shows the implementation of *sVerify* and the results on evaluating *sVerify*. Section 5 reviews related work and Section 6 concludes.

2 OVERVIEW THROUGH A MOTIVATING EXAMPLE

In this section, we first present an overview of our approach and illustrate how it works through an example.

2.1 Overview

The overall workflow of *sVerify* is shown in Figure 1. It takes as input the source code of a smart contract, including user-specified assertions, and outputs the verification result. *sVerify* first constructs a labeled control-flow graph (CFG) in which some of the nodes are labeled with assertions (if they represent program locations with assertions). Then, the labeled CFG is taken as input for the *verifier* at step 1. The verifier incrementally and systematically infers a sound invariant for each node and afterwards checks whether the invariant implies the associated assertion of the node. If yes, *sVerify* reports that the contract is verified.

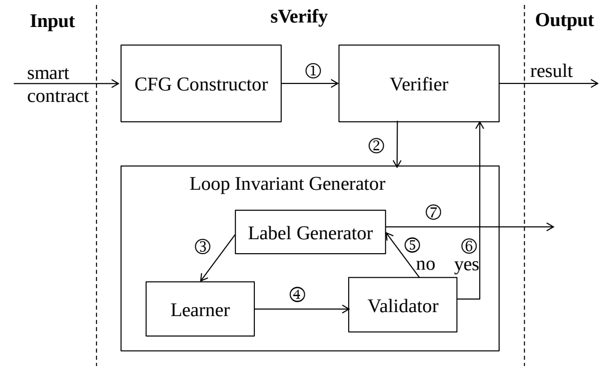


Figure 1: Overview of approach

Two methods are applied alternatively to infer invariants systematically. First, given a node *n*, we infer the invariant by computing the strongest postcondition based on the invariants associated with nodes preceding node *n*. Second, if a node *n* is a part of a loop (either explicitly due to *for* or *while* statements or implicitly due to fallback function calls), inferring the invariant is complicated. *sVerify* implements a built-in *loop invariant generator* to derive the loop invariant at step 2. The *loop invariant generator* consists of three components. A *label generator* executes the loop path with the concrete variable valuations and labels these valuations as either negative or positive samples (according to whether they satisfy the loop invariant to learn, which will be explained in Section 3.2). A *learner* conjectures a candidate invariant based on the labeled samples at step 3 through classification. A *validator* checks whether the learned candidate from *learner* at step 4 satisfies the assertion properties. If not, a counterexample is fed into the *label generator* at step 5, and this new counterexample will also be added to the data sample set for labeling. Further, a new candidate invariant is learned. This refining process will not stop until a valid invariant is checked successfully by the *validator*. This valid invariant is then returned to the *verifier* at step 6. During the data labeling phase in *label generator*, if a data sample is labeled as an error counterexample which violates an assertion property, the verification process will terminate immediately, this error counterexample is returned to the user at step 7 lastly and the contract is thus not correct.

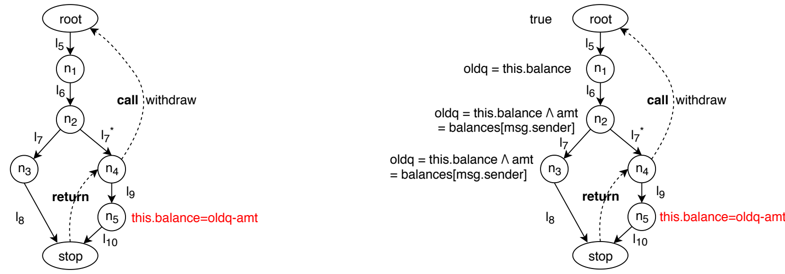
2.2 An Illustrative Example

We illustrate how *sVerify* works through an example. Figure 2 shows two versions of a smart contract which contains inter-contract function calls. In the following, we first explain the contracts in detail and then illustrate how *sVerify* detects the vulnerabilities or proves the correctness step by step.

Example. The contract shown in Figure 2a 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

```
1  contract toyDAO_A{
2  mapping (address => uint) public balances;
3
4  function withdraw() public {
5    uint oldq = this.balance;
6    uint amt = balances[msg.sender];
7    if(!msg.sender.call.value(amt)())
8      throw;
9    balances[msg.sender] = 0;
10   assert(this.balance == oldq - amt);
11 }
12 }
```

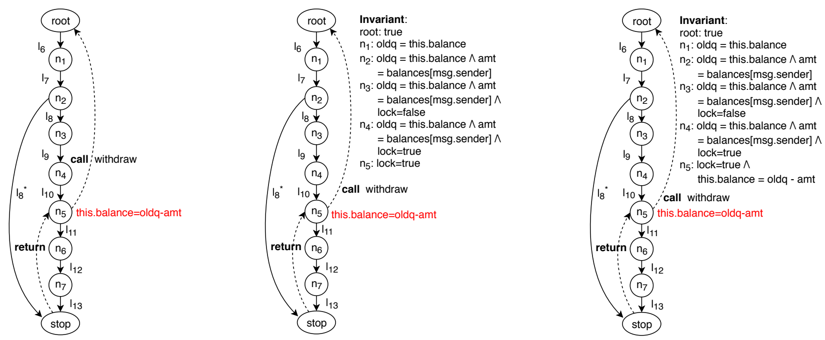
(a) source code



(b) original labeled CFG **(c) labeled CFG (v1)**

```
1  contract toyDAO_B{
2  mapping (address => uint) public balances;
3  bool locked = false;
4
5  function withdraw() public {
6    uint oldq = this.balance;
7    uint amt = balances[msg.sender];
8    require (!locked);
9    locked = true;
10   msg.sender.call.value(amt)();
11   assert(this.balance == oldq - amt);
12   balances[msg.sender] = 0;
13   locked = false;
14 }
15 }
```

(d) correct source code



(e) original labeled CFG **(f) labeled CFG (v1)** **(g) labeled CFG (v2)**

Figure 2: Contracts containing inter-contract function calls

this malicious contract is crafted to call back to the `withdraw` function again. Note that the fallback function is invoked automatically when some Ether is transferred into the contract (triggered by line 7) according to the mechanism of EVM. This action allows the attacker to claim more Ether than he deserves. The assertion at line 10 which requires the balance of the contract being decreased by `amt` exactly after line 9 will be violated in such cases. This vulnerability is also referred to as re-entrancy [3]. To prevent this vulnerability, the contract `toyDAO_B` in Figure 2d introduces a variable `lock` to ensure the transfer at line 10 can be executed only once. In addition, it should be noted that variable `lock` can only be modified by the function `withdraw`. The statement at line 8 requires the value of `lock` to be `false`, and only if this condition is satisfied, `lock` is updated to be `true` and `amt` amount is sent to the investor `msg.sender`. If there is a callback action again, the condition at line 8 fails, thus line 10 is prevented from being executed again. As a result, the assertion at line 11 is always satisfied.

To verify the `toyDAO_A` contract, *sVerify* first constructs the CFG of the `withdraw` function as shown in Figure 2b. In this CFG, node *root* and node *stop* represent the entry and exit of the function respectively. Note that in this example, *implicit edges* are introduced (systematically) to capture control flow due to the inter-contract function calls. There are two implicit edges in Figure 2b. These two edges link node n_4 to node *root* and node *stop* to node n_4 with dashed lines, which capture an inter-contract function call to the function `withdraw`. Node n_5 before an assertion statement at line 10 is called 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 2c shows the invariants of node n_1-n_3 with *root* node being true. Taking node n_2 as an example, its invariant is then strengthened based on the invariant associated with node n_1 and statement at l_6. 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 command **call** *withdraw* and ended with an edge labeled with **return** command, *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`), and then categorizes the valuations. After that it calls the learner to generate a candidate invariant which is then validated by the validator. 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, a counterexample (`amt=1, oldq=257, this.balance=256`) is generated. With this counterexample, the `msg.sender` will receive 1 wei (the smallest denomination of Ether) at line 7, and it is possible that the call back to this function gets another 1 wei. While the second call satisfies the assertion at line 10 (`amt=1, oldq=256, balance=255`), the first call which completes subsequently violates the assertion (`amt=1, oldq=257, balance=255`). After that step, the verification terminates and the contract is falsified.

For the fixed contract `toyDAO_B` in Figure 2d, the CFG of the `withdraw` function is shown in Figure 2e 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* module to generate the invariant for node n_5. Figure 2f shows the CFG with the invariants for some nodes. 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 the same node. Note that it is guaranteed that an invariant inferred at a node is indeed an invariant.

3 OUR APPROACH

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

3.1 Formalization of Smart Contracts

In this work, we focus on verifying smart contracts at the function-level. The reason is that unlike traditional programs which have *main*() function as the single entry, every function in a smart contract may be called through a transaction once the contract is initialized. Thus it is important that each function is verified separately. Without loss of generality, we define the following which captures a core set of smart contract instructions.

Definition 3.1 (Command). A command which captures the basic operation in a smart contract is defined as following.

$$\begin{array}{lcl} Com & ::= & SSTORE(p,v) \mid SLOAD(p) \mid x := Exp \mid \textbf{if } b \mid \\ & & \textbf{assert } b \mid \textbf{call } f \mid \textbf{return} \\ Exp & ::= & x \mid v \mid op(x,x) \\ op & ::= & ADD \mid MUL \mid SUB \mid DIV \mid MOD \\ b & ::= & true \mid false \mid ISZERO(Exp) \mid cmp(Exp,Exp) \mid \\ & & NOT\ b \mid b\ AND\ b \mid b\ OR\ b \\ cmp & ::= & LT \mid GT \mid EQ \end{array}$$

$SSTORE(p,v)$ writes a position p with value v to storage, while $SLOAD(p)$ reads a value of p from storage. $x := Exp$ assigns the valuation of expression Exp to variable x. The expression Exp can be a variable, 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 calling to function f and a return to the caller respectively.

Definition 3.2 (Functions of Smart Contract). A function of a smart contract 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 3.1; $\mathcal{I} : N \rightarrow Pred$ is a function that labels each node with an invariant; and $\mathcal{A} : N \rightarrow Pred$ is a function which labels each node with an assertion.

Note that 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 Ethereum Virtual Machine (EVM) bytecode with the Solidity compiler and subsequently disassemble the bytecode into EVM opcodes. The CFG is then constructed through simulating the stack based on the EVM opcode, i.e., to figure out the target of `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 of this function to the call node. Readers are referred to [9] for details on how the CFG is constructed. Initially, the node labeling function $\mathcal{I}$ is defined such that $\mathcal{I}(n) = true$ for every $n \in N$. Furthermore, the node labeling 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* by Figure 2e, the invariant of node n_5 is $\mathcal{I}(n_5) = true$, and the assertion is $\mathcal{A}(n_5) = (this.balance = oldq - amt)$.

$$\text{Sstore}\quad \frac{n \xrightarrow{SSTORE(p,v)}_e n',\ V' = V[storage[p] \mapsto v]}{(n,\Gamma,V) \xrightarrow{SSTORE(p,v)}_s (n',\Gamma,V')}$$

$$\text{Sload}\quad \frac{n \xrightarrow{SLOAD(p)}_e n'}{(n,\Gamma,V) \xrightarrow{SLOAD(p)}_s (n',\Gamma,V)}$$

$$\text{Assign}\quad \frac{n \xrightarrow{x:=Exp}_e n',\ V' = V[x \mapsto eval(Exp,V)]}{(n,\Gamma,V) \xrightarrow{x:=Exp}_s (n',\Gamma,V')}$$

$$\text{Branch}\quad \frac{n \xrightarrow{\textbf{if } b}_e n',\ V' = upd(V,b)}{(n,\Gamma,V) \xrightarrow{\textbf{if } b}_s (n',\Gamma,V')}$$

$$\text{Assert}\quad \frac{n \xrightarrow{\textbf{assert } b}_e n',\ V' = upd(V,b)}{(n,\Gamma,V) \xrightarrow{\textbf{assert } b}_s (n',\Gamma,V')}$$

$$\text{Call}\quad \frac{n \xrightarrow{\textbf{call } f}_e n',\ V[bal] > 0}{(n,\Gamma,V) \xrightarrow{\textbf{call } f}_s (n',\Gamma^\frown\langle(f,V_\Gamma)\rangle,V)}$$

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

Figure 3: Execution rules, where $(n \xrightarrow{c}_e n') \in E$; and $V \oplus V'$ denotes a valuation V_0 such that $V_0(x) = V(x)$ if x is not in the domain of V' and $V_0(x) = V'(x)$ otherwise.

Definition 3.3 (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 conforming to the semantic rules defined in Figure 3.

In Figure 3, 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 Exp in the valuation V (denoted by function *eval*). The execution rule in *Sload* is similar. After the command is executed, n is moved to the next one n', and the valuation and function call stack are not changed. The execution rule in *Branch* is similar except that the variable valuation in V

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

which does not satisfy condition b is excluded in the updated V' (denoted by function *upd*). The execution rule in *Assert* is similar to the rule in *Branch*. Rule *Call* captures the execution of an inter-contract function call, when the balance of the current contract *bal* is positive, n is moved to the root node of the called function n', and function f and the valuation of the local variables V_Γ are added to the function call stack $\Gamma^\frown\langle(f, V_\Gamma)\rangle$. 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.

A path p of a function in a smart contract is a sequence of alternating nodes and commands in the form of $\langle n_0, c_0, n_1, c_1, \dots, c_n, n_{n+1}\rangle$, where $n_0 = root$ and $n_i \xrightarrow{c_i}_e n_{i+1}$ for all $0 \le i \le n$. 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 $tr = \langle s_0, c_0, s_1, c_1, \dots, c_n, s_{n+1}\rangle$, where $s_0 = init$ and $s_i \xrightarrow{c_i}_s s_{i+1}$ for all $0 \le i \le n$. 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/state conforms to the transition relation.

Definition 3.4 (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. $\pi_n(last(tr)) = n$.

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

Definition 3.5 (Contract Correctness). Given a contract C with each function $F_i = (N_i, root_i, E_i, \mathcal{I}_i, \mathcal{A}_i)$, C is correct if $\forall F_i, n_i \in N_i \bullet \mathcal{I}_i(n_i) \Rightarrow \mathcal{A}_i(n_i)$.

Based on the constructed CFG and its semantics, the verification on the correctness of the 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 adopts two ways to infer the invariants. Given a node n, we say that m is a parent of n if and only if there is a transition from m to n. The first way is to infer an invariant of node n based on the invariants of all of its parents through computing a strongest postcondition. Before presenting how the inference works, we first define how the strongest postcondition is computed.

Definition 3.6 (Strongest Postcondition). Given a command $c \in Com$ and a precondition ϕ, the strongest postcondition $sp(c, \phi)$ is defined as:

$$sp(SSTORE(p, v), \phi) = \phi \oplus (storage[p] \mapsto v)$$

$$sp(x := Exp, \phi) = \begin{cases} \phi \oplus (x \mapsto v) & \text{if } Exp = v \\ \exists y \bullet x = Exp[x \leftarrow y] \wedge \phi[x \leftarrow y] & \text{otherwise} \end{cases}$$

$$sp(c, \phi) = \phi \wedge b \qquad \text{if } c = \textbf{if } b \text{ or } \textbf{assert } b$$

$$sp(c, \phi) = \phi \qquad \text{if } c = SLOAD(x) \text{ or } \textbf{return}$$

$$sp(\textbf{call } f, \phi) = \forall x \in GV \bullet \phi \ominus \phi(x)$$

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

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

In the above definition, the predicate ϕ is overwritten by predicate $(storage[p] \mapsto v)$ for *SSTORE* command. Note that symbol $\oplus$ overwrites the predicate related to $storage[p]$[2] if it exists; otherwise, the postcondition is the conjunction of the predicate and ϕ. The strongest postcondition for assignment has two cases. One is similar to the *SSTORE* command if it assigns a value to a variable. The other is the conjunction of the precondition ϕ and the assignment predicate which rewrites the variable x in *Exp* and ϕ. For branching and assertion commands, the strongest condition is the conjunction of ϕ and condition b. For command *SLOAD* or **return**, the strongest postcondition is ϕ.

We remark that the strongest postcondition for command **call** f is ϕ except that all constraints related to global variables GV are eliminated. Note that symbol $\ominus$ represents variable elimination of all storage variables in ϕ. This is designed as a function call f could potentially modify the valuation of the storage variables by invoking other functions in the contract. 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 storage variables that are modified by each function in the contract. With that information, we strengthen the above rule as follows: all constraints on storage variables except those which are only modified in the current function are eliminated. This is sound as all callback actions to the current function is captured in the CFG.

Algorithm 1 shows details on how to update the invariant of a node based on the strongest postcondition. Let Ψ be a predicate which is initially $false$. We compute $sp(c, \mathcal{I}(m))$ for each transition (m, c, n). 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 6, 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 6 ensures that any node which has no parent node like the root node is not updated. Taking node n_2 in Figure 2b as an example, its invariant inferred through Algorithm 1 is $oldq = this.balance \wedge amt = balances[msg.sender]$ which is strengthened by executing the command at line 6 from node n_1 whose invariant is $oldq = this.balance$.

The following establishes the correctness of the above algorithm.

Proposition 3.7. *The invariant inferred through Algorithm 1 is indeed an invariant by Definition 3.4.*

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 in the form of $\langle n_0, c_0, n_1, c_1, n_2 \dots, n_k, c_k, n_0\rangle$, the invariant of node n_0 is

[2] which is implemented through variable elimination

Algorithm 2: Algorithm $generateLI(F, n)$

1 Let DS be the set of randomly generated valuations of Var at node n;
2 **while** *not timeout* **do**
3 add all valuations at node n to DS after executing the loop with any valuation from DS and label all valuations in DS;
4 **if** $checkCE(DS)$ *is not empty* **then**
5 return "falsified" with an error sample;
6 **end**
7 $\phi = learnINV(DS)$;
8 **if** $validate(\phi, F, n)$ *is true* **then** return ϕ ;
9 **else** add the counterexample generated by $learnINV(DS)$ to DS ;
10 **end**
11 return a timeout message;

recursively inferred based on itself and thus may never terminate. Therefore, we distinguish nodes which are the head nodes of certain loops (i.e., a node representing the start of a loop statement or an external function call) 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, i.e., *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 set of valuations of variables in Var at node n (denoted as DS) is initially generated through random sampling at line 1. The size of the initial DS is decided empirically, e.g., 20 samples in most cases. In general, a reasonably large set of random samples is often helpful in learning candidate invariants efficiently. Based on the CFG and the initial input which is a concrete valuation from DS, the program is executed from node n until termination. During the execution, node n may be visited again and the variable valuations upon reaching n are added to DS as well. Next, we label each variable valuation in DS into 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, it is labeled

- '+': 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 illustrative contract in Figure 2d 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 10. After executing with these valuations at line 10, 1 more valuation is added to DS: $(2, 0, 20, 16)$. Afterwards, valuation $\{(5, 1, 30, 25)\}$ is labeled with '+'; and $\{(2, 0, 20, 18), (2, 0, 20, 16)\}$ are labeled with '-'.

After categorizing the variable valuations, we check further whether there is any valuation labeled with 'e' with the function $checkCE(DS)$ in lines 4-6. If there is one, the invariant generation function returns "falsified" together with the valuation as a counterexample. Otherwise, we invoke function $learnINV$ to start the *learning* phase (line 7). The primary idea here 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 [47], one of the most efficient and powerful classification techniques that is built upon SVM and the decision tree classification, to infer a candidate invariant in the form of arbitrary combination of conjunction or disjunction of linear inequalities. The learning result is returned as the candidate invariant ϕ for further validation.

In the *validation* phase (lines 8-9), function $validate(\phi, F, n)$ is invoked to check whether the learned 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 every path which starts with n and ends with a parent of n. The invariant is inductive if and only if, for all m such that $(m, c, n) \in E$, $sp(\mathcal{I}(m), c) \Rightarrow \phi$. In *sVerify*, this is implemented through solving the satisfiability of $sp(\mathcal{I}(m), c) \wedge neg(\phi)$ using SMT solvers where neg is the negation function. If it is unsatisfiable for all m, ϕ is inductive and thus indeed an invariant. Otherwise, a counterexample in the form of variable valuation is generated and added to the data set DS 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 [47] is guaranteed to terminate given a finite set DS, the overall learning process may timeout due to too many guess-and-check iterations. In the case that there is no inductive invariant returned from Algorithm 2 after a pre-set amount of time units, we adopt the simple heuristics of treating the conjunction of the assertion with the current candidate as a candidate invariant for validation. This is justified intuitively as the learned invariant should be strong enough to imply the assertion. For example, in contract `toyDAO_B` shown in Figure 2d, a candidate invariant $lock = true$ is generated by Algorithm 2. However, a time-out occurs when *sVerify* aims to validate it. Applying the heuristics, the candidate invariant is strengthened as $lock = true \wedge this.balance = oldq - amt$, which is subsequently validated.

3.3 Overall Verification Algorithm

With the above discussion, we are now ready to present the overall algorithm which is shown in Algorithm 3. Given a smart contract C with N functions, we first construct one CFG for each function at line 1. For each node n in each function F, we update the corresponding node invariant with lines 8 and 10. Whenever the invariants stabilize (i.e., reaches a fixed point), we check whether, for every node, its invariant implies its assertion. If this implication checking fails at any node, the counterexample returned by the SMT solver

Algorithm 3: Overall Verification Algorithm

```
1  {F_1, F_2, ..., F_N} ← CFG_construction(C);
2  for F ∈ {F_1, F_2, ..., F_N} do
3      I′ ← ∅;
4      while I′ ≠ I do
5          I′ ← I;
6          for n ∈ N do
7              if n is loop head then
8                  I(n) ← generateLI(F, n);
9              else
10                 I(n) ← inferI(F, n);
11             end
12         end
13     end
14     for n ∈ N do
15         if I(n) ⇏ A(n) then
16             CE ← counterexample;
17             break if CE is checked to be an actual
               counterexample;
18         end
19     end
20 end
21 return verification succeeds
```

is checked to see whether it is an actual counterexample (through symbolic execution). If it is, the counterexample is returned as evidence to falsify the contract. If all assertions are implied by the invariants at the corresponding nodes, the contract is successfully verified.

For instance, consider the function $withdraw()$ in the contract shown in Figure 2d. Initially the invariant of each node is *true*, and then the algorithm incrementally strengthens the node invariants. In this example, the invariants of all nodes except node n_5 are updated by function $inferI(F, n)$. The invariant of node n_5 which is head node of the loop is updated by function $generateLI(F, n)$. Afterwards, the algorithm checks whether the invariant of each node implies its associated assertion. In this example, the only assertion which is not *true* is at node n_5. The invariant of node n_5 ($locked = true \wedge this.balance = oldq - amt$) shown in Figure 2g implies its assertion ($this.balance = oldq - amt$). As a result, the algorithm concludes at line 21 that verification succeeds.

Theorem 3.8. *Algorithm 3 is sound.*

Proof. There are two verification results by Algorithm 3. Either the contract is falsified or verified. In the former case, the result is sound as a counterexample is returned only if it is validated to be an actual counterexample. In the latter case, the soundness is established on 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 3.7. In the latter case, the correctness of the inferred invariant generated by function *generateLI* 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' only when all assertions are implied by the invariants (by Definition 3.5). □

Algorithm 3 is not always terminating as the loop invariant generation method *generateLI* is not always terminating. Note that if function *generateLI* always terminates, Algorithm 3 always terminates. In our implementation, a time limit is applied to the function and thus Algorithm 3 always terminates. The exact complexity of the overall verification algorithm is hard to analyze due to the many components that it depends on. We thus evaluate it empirically in the next section.

4 IMPLEMENTATION AND EVALUATION

sVerify is based on a symbolic execution engine sCompile [9], which was developed for smart contracts written in Solidity. *sVerify* first compiles an Ethereum smart contract into EVM bytecode with the Solidity compiler and subsequently disassembles the bytecode into EVM opcodes with the toolkit provided in EVM for constructing the CFG. To learn a candidate invariant, *sVerify* implements the Linear-Arbitrary algorithm based on LIBSVM [8] and C5.0 [40]. During the candidate validation phase, *sVerify* adopts Z3 SMT solver [13] to check the satisfiability of the conditions. There is also a built-in module in *sVerify* to automatically generate assertions for capturing common code vulnerabilities such as arithmetic overflow. For example, given a transition $(n, (c = a - b), n')$ where the types of variables are all `uint256`, *sVerify* automatically updates the assertion function of node n to be $\mathcal{A}(n) = \mathcal{A}(n) \wedge (b \leq a)$.

In the following, we conduct two sets of experiments to evaluate *sVerify* on real-world smart contracts. We focus on the following questions.

(1) How effective is *sVerify* in verifying smart contracts against common code vulnerabilities?
(2) How effective is *sVerify* in verifying contract-specific assertions?
(3) How efficient is *sVerify*?

All experiments are run on a 64-bit machine having Intel i7-7500U CPU at 2.7GHz with 4 cores and 16 GB of RAM, running Ubuntu 18.04LTS. As of now, *sVerify* is developed for Solidity version 0.4.25 and Ethereum 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, 93.3% (476/510) of the vulnerabilities reported in the CVE list [12] between 2018 and 2019 are overflow. The DAO attack [1], one of the most famous attacks which caused huge monetary loss, has evidenced the importance of verifying smart contracts against 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 Section 2.2.

For baseline comparison, we focus on two state-of-the-art verification tools Zeus and SOLC-VERIFY. Zeus [24] is a framework for automatic formal verification of smart contracts based on abstract interpretation techniques. SOLC-VERIFY [20] is a tool that allows specification and modular verification of Solidity smart contracts which is built upon the Boogie verifier [5].

Setup. To compare with Zeus, we adopt the test subjects reportedly analyzed by Zeus in [24] and systematically run *sVerify* on them. 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[3]. 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, we further ignore 61 smart contracts. The remaining 835 contracts are taken as the test subjects.

To compare with SOLC-VERIFY, we evaluate *sVerify* with the same subjects as SOLC-VERIFY, which consists of 3897 contracts that can be successfully evaluated by SOLC-VERIFY among 7836 contracts. SOLC-VERIFY is configured with v0.4.25-boogie. The flag for arithmetic is set to be mod-overflow and the other options are as default. Timeout for each contract is 2 hours for both SOLC-VERIFY and *sVerify*. Furthermore, a 10 seconds timeout is set for each Z3 solver request.

Results. The experiment results on Zeus's 835 test subjects are summarized in Table 1, where column *Finished* shows the number of smart contracts successfully analyzed by these three tools. The comparison is conducted based on four criteria, i.e., True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN). Column *Correct* is the percentage of correct results, i.e., TP and TN to the total finished contracts.

We have multiple observations based on the results. First, SOLC-VERIFY only finishes 56 contracts among the 835 contracts, which is significantly fewer than that of *sVerify* and Zeus. Besides the Solidity version problem, it is mainly due to the limited features that SOLC-VERIFY supports. Second, compared with Zeus, *sVerify*'s verification results are more reliable since there are fewer false positives. In particular, for overflow, Zeus generates 30 false positives and 2 false negatives, whereas *sVerify* has only 2 false positives and 0 false negative; for re-entrancy, *sVerify* has 1 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 Figure 4 in the following which may offer clues.

Zeus generates a false alarm of overflow for function `split()`[4], which sends tokens to two accounts. We speculate the false alarm is due to line 4, 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()`[5]. At line 8, expression

```
1  function split() payable public {
2    uint fee = msg.value / 100;
3    feeRecipient.send(fee);
4    etcDestination.call.value(msg.value - fee)();
5  }
6  function process(bytes32 _destination) payable returns (bool) {
7   if (msg.value < 100) throw;
8   var tax = msg.value * taxPerc / 100; ...
9  }
10 function testNumberRequest(address randomreality, ...) payable {
11    RandomRealityAPI randomrealityapi = RandomRealityAPI(
          randomreality);
12    uint256 cost = randomrealityapi.getPrice(200000);
13    bytes32 id = randomrealityapi.requestNumber.value(cost)(...);
          ...
14 }
15 function transferFrom() returns (bool success) {
16    ...
17    if(now < startTime + 1 years) ...
18 }
19 function multisend(..., address[] dests, uint256[] values){
20    uint i = 0;
21    while (i < dests.length) {
22       ERC20(_tokenAddr).transfer(dests[i], values[i]);
23       i += 1;
24    } ...
25 }
```

Figure 4: Example functions incorrectly analyzed by Zeus, *sVerify* or SOLC-VERIFY

`msg.value*taxPerc/100` may exceed the maximum value. For re-entrancy, one example case Zeus misses is the vulnerability in function `testNumberRequest`[6]. Attackers may input some address to exploit the re-entrancy vulnerability at line 13.

The reason of two false positives generated 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()`[7] is only set in the constructor and the overflow at line 17 is in fact impossible.

For test subjects from SOLC-VERIFY, out of the 3897 contracts, *sVerify* identifies 1236 vulnerable contracts and times out on 38 contracts. Note that most of the unfinished contracts are caused by the bulk of shared modules. In comparison, SOLC-VERIFY identifies 438 contracts. Given the large number of vulnerable contracts, we focus on a set of 68 contracts which have at least 100 transactions and manually examine them. Note that it is also the same set discussed by SOLC-VERIFY in [20]. The results are shown in Table 2[8]. It can be observed that SOLC-VERIFY has more false positives compared to *sVerify*. There are multiple reasons why false alarms may be generated by SOLC-VERIFY. For instance, missing range assumptions for array lengths cause false overflow alarms for loop counters. Readers are referred to [20] for details. Furthermore, *sVerify* identifies more true positives. One example is the function `multisend` shown in Figure 4. SOLC-VERIFY reports `i+=1` might overflow, which is regarded as a false alarm. However, *sVerify* reports the index of variable `values` at line 22 might cause overflow if `i` is larger than the length of array `values`, which is a true positive that is missed

[3] at https://etherscan.io/ and https://www.etherchain.org/. About 450 contract source codes from website etherCamp (https://live.ether.camp/) are not available any more, and about 120 contract addresses are not correct like missing some letter or only listing part of the address as "Code_3_fdf6d_faucet"

[4] contract address: 0xc8D9890df1ff2E87BE05e9EDaB3ccA26F054b611

[5] contract address: 0x9053d234a1ff2290f087a1ca9460e3263121e580

[6] contract address: 0x0d54292b728730f563fc1eb1b2cbbce79bc1dbcf

[7] contract address: 0x08711D3B02C8758F2FB3ab4e80228418a7F8e39c

[8] Some of the overflow identified by *sVerify* are the results of some problematic compiler induced checking, which have now been fixed in the latest Solidity compiler.

Table 1: Comparison results on Overflow and Re-entrancy with Zeus and SOLC-VERIFY.

	SOLC-VERIFY						Zeus						*sVerify*					
Category	Finished	TP	TN	FP	FN	Correct	Finished	TP	TN	FP	FN	Correct	Finished	TP	TN	FP	FN	Correct
Overflow	56	12	44	0	0	100%	826	562	232	30	2	96.1%	804	548	254	2	0	99.8%
Re-entrancy	56	3	53	0	0	100%	831	19	782	9	21	96.4%	804	43	761	0	0	100%

Table 2: Comparison results on Overflow between SOLC-VERIFY and *sVerify*.

	SOLC-VERIFY						*sVerify*					
Category	Finished	TP	TN	FP	FN	Correct	Finished	TP	TN	FP	FN	Correct
Overflow	68	4	35	29	0	57.4%	68	17	43	8	0	88.2%

by SOLC-VERIFY. Note that there are also 8 false alarms generated by *sVerify*. Besides the missing constraints on time like the case of aforementioned function `transferFrom()`, the balance of Ether also matters. For instance, *sVerify* generates an overflow alarm for a statement like `uint amt=msg.value*20000`. This multiplication is regarded as a false alarm as the current amount of total Ether is limited and thus the result `amt` does not overflow.

Efficiency of sVerify. sVerify successfully analyzed 804 (out of 835) contracts and 3859 (out of 3897) contracts for two sets of benchmarks (with a timeout of two hours), and each contract takes an average of 38.5s and 14.8s 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 1.24s. The reason why *sVerify* takes a longer time comparing to Zeus and SOLC-VERIFY is that, *sVerify* needs to learn the invariant for loops in the verification process, which is an essential step to acquire an accurate result. Overall, we believe that *sVerify* verifies or falsifies smart contracts in a reasonable time.

4.2 Verifying Contract-specific Assertions

While verification against common vulnerabilities is important, it is far from sufficient in verifying the functional correctness of smart contracts. In the second set of experiments, we identify multiple 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, i.e., they often require learning the relevant loop invariants. As most of the loops operate on arrays, we define several patterns to the properties of arrays, e.g., `assert(ret==ARRAY_MAX)` to check whether the returned value `ret` by the program is the maximum value of the array.

The test subjects consist of 7 representative contracts from accounts that rank top 1000 in terms of balances, including the prevailing *multiSig* wallet, several token issuing contracts and some decentralized exchange contracts. These 7 contracts are particularly important as many other contracts are built upon them. Table 3 shows the verification results by *sVerify* and SOLC-VERIFY, where column *#loc* stands for the lines of code, *#pubfns* for the number of public functions, and *#lpfns* for the number of functions with loops. A 'NULL' result means unfinished analysis caused by exception.

```
1  function getTransactionIds(uint from, uint to, ...) public
2     returns (uint[] _transactionIds){
3     uint[] memory transactionIdsTemp=new uint[](transactionCount);
4     ...
5     _transactionIds = new uint[](to - from);
6     ...
7  }
8
9  function withdraw(uint _amount){
10    ...
11    tokens[0][msg.sender] = safeSub(tokens[0][msg.sender], amount);
12    if (!msg.sender.call.value(amount)())
13    ...
14    assert(this.balance == oldq - amt);
15 }
```

Figure 5: Alarm functions by *sVerify*

sVerify successfully finishes analyzing all the contracts whereas SOLC-VERIFY finishes 3. *sVerify* reports 4 alarms, where 2 alarms are actual vulnerabilities. In the function `getTransactionIds` of contract `MultiSigWallet`[9] shown in Figure 5, the statement at line 5 overflows if the assigned value of variable `to` is smaller than `from` (which will spawn a bunch of new arrays and eventually cost up all the gas of the transaction). The other one is in the function `withdraw` of contract `TokenStore`[10], the assertion statement at line 14 is violated if the fallback function in `msg.sender` calls back to function `withdraw` again. However, in practical runtime, the smart contract developer decreases the token amount of the `msg.sender` at line 11, so it is impossible for the `msg.sender` to claim more Ether than he deserves. The other two false alarms are due to limitations on analyzing functions in isolation, as explained the constraints of time and Ether balance in Section 4.1. Two overflow false alarms are all eliminated after inserting requirements for capturing the arithmetic overflow. In comparison, SOLC-VERIFY reports 5 alarms which are all false alarms, and 3 overflow alarms are also eliminated after the same insertion.

This test shows that *sVerify* can verify the contract-specific assertions accurately.

[9] 0x231568bAA78111377F097bB087241F8379fa18f4
[10] 0x1cE7AE555139c5EF5A57CC8d814a867ee6Ee33D8

Table 3: Real-world Contracts Analysis.

Contract	Description	#loc	#pubfns	#lpfns	*sVerify*		SOLC-VERIFY	
					overflow	re-entrancy	overflow	re-entrancy
MultisigWallet	Wallet controlled by multiple owners to transfer ETH	304	14	7	TP	TN	NULL	NULL
Imt	Wallet controlled by single to receive and transfer ETH	65	4	1	TN	TN	FP	TN
WithdrawDAO	Withdraw investment from DAO for investors	15	2	0	FP	TN	FP	FP
LifCrowdsale	Token issuance and crowsale	800	37	1	TN	TN	NULL	NULL
WETH	Convert ETH to same amount of WETH token	50	6	0	FP	TN	FP	FP
KyberReserve	Decentralized, p2p crypto asset exchange	298	19	2	TN	TN	NULL	NULL
TokenStore	Decentralized exchange for ERC-20 tokens	240	20	3	TN	TP	NULL	NULL

4.3 Threat to Validity

One of the threats to validity is the selection of benchmark smart contracts. As we intend to implement a smart contract verification tool based on the user-provided assertions, finding a suitable smart contract benchmark suite is essential to validate and evaluate the implementation. The range of suitable benchmark suite is further restricted as we must compare with other existing verification tools. We are thus restricted to two common code vulnerabilities (which are supported by multiple tools that we can compare with) as well as a relatively small set of contracts so that we can manually specify the assertions and check whether the verification results are correct.

5 RELATED WORK

Many approaches have been proposed recently to test or verify smart contracts through various techniques. For instance, the fuzzing tools reported in [2, 23, 46] 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. The other works adopt the symbolic technique to analyze the correctness of smart contracts [11, 29, 36, 38]. However, symbolic execution usually suffers from path explosion. As a result, these approaches are forced to bound the search space by, for instance, setting a limit on the number of jumps or function calls. These approaches are thus designed for testing smart contracts rather than verifying them.

Unlike these approaches, Securify [43] is a static analyzing tool based on abstract interpretation and dependency graph. It produces vulnerability patterns through inference rule-based generation and analyzes the correctness accordingly. Securify suffers from the limitation of abstraction interpretations such as fixed abstraction domains and restriction on manually defined semantic properties. In addition, it cannot verify numerical properties like overflow. VerX [39] introduces delayed predicate abstraction approach based upon symbolic execution and abstraction techniques to verify real-world smart contracts during transaction execution. However, VerX only supports the external-call-free contracts whose behavior is equivalent to the behavior of the contracts without callbacks.

There are some approaches which translate smart contracts into different intermediate representations and then utilize existing analysis tools of the intermediate representations to analyze smart contracts. Bhargavan *et al.* [6] translated smart contracts into F* programs and thus the contract verification problems are reduced to F* program verification problems. Unfortunately, this approach is unsound and incomplete as it leaves out loops during the translation. Zeus [24], proposed by Kalra *et al.*, translates smart contracts into LLVM bitcode and leverages Seahorn [18] as the symbolic model checking backend to reason about contract correctness. SOLC-VERIFY [20] and verisol [44] 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 source code. However, since Boogie was not designed from smart contracts, some features are not supported for the translation. Another interesting but slightly different work is done by VeriSolid [31], which extends FSolidM [30] to model Ethereum smart contracts as transition systems with guarded transitions. However, VeriSolid is designed for a different purpose from *sVerify*. It aims to generate correct-by-construction smart contract code from abstract formal models.

There are also some approaches on verifying smart contracts using the theorem proving approach. Hildenbrandt *et al.* defined a formal semantics of the EVM using the K framework [41], and KEVM provides a basis for theorem proving Ethereum smart contracts. Hirai [21] defined EVM in Lem [33], and proved some safety properties of Ethereum smart contracts in Isabelle/HOL [37]. However, verification through theorem proving is not fully automated and requires manual efforts.

In our work, we propose an approach for verifying smart contracts based on lazy annotation and automatic loop invariant generation. Recently, a number of loop invariant generation approaches have been proposed. These include those based on abstraction interpretation [15, 26], counterexample-guided abstraction refinement [4, 10] or interpolation [22, 28], constraint solving and logical inference [14, 17], and learning [16, 27, 47]. The former three approaches depend on constraint solving and thus suffer from scalability. Unlike these approaches, we adopt the learning-based loop 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 effectively verify or falsify the popular smart contracts. We evaluated *sVerify* on real-world smart contracts and the results show that *sVerify* is effective and reasonably efficient. In the future, we plan to extend our work to contract-level invariants.

REFERENCES

[1] 2016. The DAO Attacked: Code Issue Leads to 60 Million Ether Theft. https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft.

[2] Sefa Akca, Ajitha Rajan, and Chao Peng. 2019. SolAnalyser: A Framework for Analysing and Testing Smart Contracts. 482–489. https://doi.org/10.1109/APSEC48747.2019.00071

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts SoK. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. Springer-Verlag New York, Inc., 164–186.

[4] Thomas Ball and Sriram K. Rajamani. 2001. The SLAM Toolkit. In *Computer Aided Verification*. 260–264.

[5] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. 2005. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*. Springer, 364–387.

[6] 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 '16)*. ACM, 91–96.

[7] Jerry Brito and Andrea Castillo. 2013. *Bitcoin: A primer for policymakers*. Mercatus Center at George Mason University.

[8] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* 2 (2011), 27:1–27:27. Issue 3. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[9] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, Yan Cai, and Zijiang Yang. 2019. sCompile: Critical Path Identification and Analysis for Smart Contracts. In *Proceedings of the 21st International Conference on Formal Engineering Methods, ICFEM 2019*. 286–304.

[10] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *J. ACM* 50, 5 (2003), 752–794.

[11] ConsenSys. 2018. MyThril: Security analysis of Ethereum smart contracts. https://github.com/ConsenSys/mythril. [online, accessed 30-may-2019].

[12] CVE. [n.d.]. CVE List. https://cve.mitre.org/data/downloads/index.html. (Accessed on 12/24/2019).

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, 337–340.

[14] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive Invariant Generation via Abductive Inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. 443–456.

[15] Cormac Flanagan and Shaz Qadeer. 2002. Predicate Abstraction for Software Verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, 191–202.

[16] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification*. 69–87.

[17] Ashutosh Gupta and Andrey Rybalchenko. 2009. InvGen: An Efficient Invariant Generator. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*. 634–640.

[18] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Computer Aided Verification*. Springer, 343–361.

[19] Stuart Haber and W Scott Stornetta. 1990. How to time-stamp a digital document. In *Conference on the Theory and Application of Cryptography*. Springer, 437–455.

[20] Ákos Hajdu and Dejan Jovanovic. 2019. solc-verify: A Modular Verifier for Solidity Smart Contracts. *CoRR* abs/1907.04262 (2019). arXiv:1907.04262 http://arxiv.org/abs/1907.04262

[21] Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Financial Cryptography and Data Security*. 520–535.

[22] Joxan Jaffar, Jorge A. Navas, and Andrew E. Santosa. 2012. Unbounded Symbolic Execution for Program Verification. In *Proceedings of the Second International Conference on Runtime Verification (RV'11)*. 396–411.

[23] Bo Jiang, Ye Liu, and W. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. 259–269. https://doi.org/10.1145/3238147.3238177

[24] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *NDSS*. The Internet Society.

[25] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[26] Vincent Laviron and Francesco Logozzo. 2009. SubPolyhedra: A (More) Scalable Approach to Infer Linear Inequalities. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 229–244.

[27] Jiaying Li, Jun Sun, Li Li, Quang Loc Le, and Shang-Wei Lin. 2017. Automatic loop-invariant generation and refinement through selective sampling. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. 782–792.

[28] S. Lin, J. Sun, T. K. Nguyen, Y. Liu, and J. S. Dong. 2015. Interpolation Guided Compositional Verification (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 65–74.

[29] 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 '16)*. ACM, 254–269.

[30] Anastasia Mavridou and Aron Laszka. 2018. Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts. (02 2018).

[31] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. 2019. *VeriSolid: Correct-by-Design Smart Contracts for Ethereum*. 446–465. https://doi.org/10.1007/978-3-030-32101-7_27

[32] Kenneth L. McMillan. 2010. Lazy Annotation for Program Testing and Verification. In *Proceedings of the 22Nd International Conference on Computer Aided Verification (CAV'10)*. 104–118.

[33] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. 175–188.

[34] Satoshi Nakamoto. 2019. *Bitcoin: A peer-to-peer electronic cash system*. Technical Report. Manubot.

[35] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. 2016. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press.

[36] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018*. ACM, 653–663.

[37] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer.

[38] Trail of Bits. 2018. Manticore: Symbolic Execution Tool. https://github.com/trailofbits/manticore. [online, accessed 30-may-2019].

[39] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contract. In *IEEE Symposium on Security and Privacy*.

[40] J.R. Quinlan. 2017. C5.0: An Informal Tutorial. http://www.rulequest.com/see5-unix.html.

[41] Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.

[42] Nick Szabo. 1997. The idea of smart contracts. http://szabo.best.vwh.net/smart_contracts_idea.html.

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

[44] Yuepeng Wang, Shuvendu Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. 2019. Formal Specification and Verification of Smart Contracts for Azure Blockchain. (April 2019). https://www.microsoft.com/en-us/research/publication/formal-specification-and-verification-of-smart-contracts-for-azure-blockchain/

[45] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum Project Yellow Paper* 151 (2014), 1–32.

[46] Valentin Wüstholz and Maria Christakis. 2019. Harvey: A greybox fuzzer for smart contracts. *arXiv preprint arXiv:1905.06944* (2019).

[47] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A Data-driven CHC Solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, 707–721.

[48] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan Bach D Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2019. Smart Contract Development: Challenges and Opportunities. *IEEE Transactions on Software Engineering* PP (09 2019), 1–1. https://doi.org/10.1109/TSE.2019.2942301