Verifying Smart Contracts by Learning Contract Invariants

Bo Gao, Ling Shi, Jiaying Li, and Jun Sun

Abstract—Smart contracts are computer programs run on blockchain which allow different parties to conduct safe transactions. They are immutable after deployment and a minor flaw may result in huge financial losses. Thus, the safety and security of smart contracts are critical for the developers and users. Early efforts focused on verifying smart contracts at the function level. Although a few recent approaches aim to verify smart contracts at the contract level (which is more useful), adopting them for contract analysis requires non-trivial efforts as they often either require user-provided contract invariants as inputs or rely on limited invariant templates, which impedes their application in practice.

In this work, we develop an approach which verifies contract-level correctness through learning contract invariants. In particular, we develop *XVerify*, a formal verifier for Solidity smart contracts, with the capability of counter-example guided contract invariant learning. *XVerify* has been evaluated on 87 real-world popular smart contracts against more than 12k assertions on multiple critical properties. The experimental result shows *XVerify* outperforms existing smart contract verification approaches in terms of effectiveness and efficiency.

Index Terms—Smart Contract, Verification, Contract Invariants

1 Introduction

Smart contracts, running on blockchain platforms like Ethereum [1], EOS [2] and Hyperledger Fabric [3], are computer programs which allow users to define complex protocols among distrusting parties. With the help of smart contracts, blockchain is widely used in many different areas such as supply chain management, financial services and data sharing [4], [5], [6]. Typically, a smart contract is written in high-level languages, like Solidity [7], Vype [8] in Ethereum. The high-level contracts are then compiled into Ethereum Virtual Machine (EVM) [1] bytecode, which are deployed to blockchain later on. Different from traditional programs, smart contracts cannot be modified or patched once deployed on the blockchain. That is called immutability, one of key features of blockchain. At the same time, smart contracts usually manage a significant amount of financial assets, such as cryptocurrencies and tokens. Any defect makes the contract forever vulnerable and may lead to great financial losses. For instance, the notorious Parity incidents [9], [10] caused a loss of more than 150k Ether ($\sim$ 30M USD) during the first attack, and freezed about 500k Ether ($\sim$ 150M USD) during the second attack. Note that Ether is the native cryptocurrency of Ethereum. Thus, formal verification of smart contracts is highly desirable.

Several tools have been developed for the verification of smart contracts in recent years. Securify [11] and Zeus [12] encode Solidity contracts into existing intermediate languages (i.e., Datalog and LLVM) and reuse existing verification facilities to analyze the contracts against predefined security properties. However, these tools focus on the function-level analysis of smart contracts, which often lead to false alarms. On the other hand, VerX [13] uses a delayed predicate abstraction approach to verify user-provided functional specifications by automatically inferring the abstraction predicates from contract codes or atomic properties. Such an approach is idealistic as providing a specification can be a burden for users. solc-verify [14] translates Solidity contracts into the Boogie language and discharges verification conditions with SMT solvers. It also requires users to provide different annotations for accurate verification within the contract source codes. Solicitous [15], the formal engine inside the official Solidity compiler, directly models the Solidity contracts with constrained Horn clauses and leverages a generic theorem provers for fully contract verification. Its capability is greatly dependent on the CHC provers. VeriSmart [16] automatically discovers transaction and loop invariants with the help of domain-specific templates to verify smart contracts. However, they are limited to a few simple forms, and many forms of useful invariants are not taken into consideration, such as $x = y + z$. This limitation constrains the capabilities of the tool.

In this work, we present the design and implementation of a formal verifier called *XVerify* for EVM contracts. *XVerify* verifies unbounded contract-level correctness through learning contract invariants based on a combination of symbolic execution [17], lazy annotation [18] and state-of-the-art loop invariant generation [19], [20]. Given a smart contract with assertions (represented by opcode 0xfe in EVM bytecode), an accurate control-flow graph (CFG) is constructed first. Then, *XVerify* leverages symbolic execution [17] to generate verification conditions which form the node invariants for each node in the CFG. The invariants (which is $true$ initially) are then monotonically strengthened

- *B. Gao is with Singapore University of Technology and Design, Singapore. E-mail: bo_gao@mymail.sutd.edu.sg*
- *L. Shi is with Nanyang Technological University, Singapore. J. Li is with Microsoft, China. J. Sun is with Singapore Management University, Singapore.*

```
function distribute(address[] investor) external {
    for(uint i=0; i < investor.length; i++)
        investor[i].transfer(10);
}
```

Fig. 1. Bytecode vulnerability

```
contract GRX {
    uint buyPrice = 760;
    function buy() payable public returns (uint amount){
        amount = msg.value * buyPrice;
        assert (msg.value == amount/buyPrice);
        ...
        return amount;
    }
}
```

Fig. 2. Example contract adapted from real-world contract

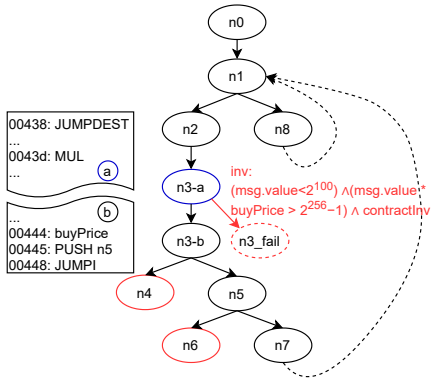

Fig. 3. Example CFG for GRX contract

through sound inference rules. Next, *XVerify* aims to identify the contract invariants and loop invariants, if there are loops in the contracts, with the help of counter-example guided invariant learning. Contract invariants are properties that hold through any function call sequences on the contract, which are essential for the verification of contract correctness. Note that learning contract invariants is slightly different from learning ordinary loop invariants (for reasons such as: there is no terminating condition). Domain-specific heuristics are also used to facilitate the verification.

Compared with existing approaches, our approach has the following advantages. First, our approach requires no user-specified contract invariants. Specifying invariant is often overwhelming for ordinary users and such a requirement would hinder the application of an approach. To address this issue, our approach learns invariants automatically in a property-guided manner. Second, our approach is based on EVM bytecode which is more accurate comparing to those based on source code. One of such examples is shown in Figure 1, users can invoke function *distribute* to pay Ether back to the investors. The input data to invoke the function is comprised of the function selector, the offset of the array data, the length and the content of the array. Interested readers can refer to Solidity documentation [7] for further details on the components of the input data like the function selector. Early Solidity versions, before v0.5.0 as far as we know, did not have any check when arithmetic computations involved the offset of the array data. It is thus possible for malicious users to craft the input data to distort the function, like transfer money to the same address. Such compiler-related deficits are blind spots for source-code based tools. Third, our approach is more efficient. We adopt the idea of lazy annotation to traverse back from the fail nodes to the root node on the control flow graph (CFG) to get the verification conditions, which means we concentrate on the property-related nodes only. Minimizing the number of queries to the SMT solver saves a lot of time and improves the efficiency.

Experiments demonstrate the effectiveness and efficiency of *XVerify* for analyzing smart contracts. We first conducted a comparison experiment with Solicitous [15] on 30 contracts with version v0.6. Then we further evaluated *XVerify* with VeriSmart [16] on 57 solidity contracts (top30 transactions and top30 balance) whose versions are v0.5. The result showed that *XVerify* can verify more contracts and detect more vulnerabilities against assertions (including compiler inserted assertions) within shorter analysis time than the other two tools.

To conclude, our main contributions in this work include the following.

- We develop a new approach for verifying smart contracts. Our approach infers contract invariants with the help of loop invariant learning and invariant inference techniques to accurately verify the smart contracts.
- We implement an end-to-end tool *XVerify* for EVM smart contracts and make it public on Github (it can be found at: https://www.dropbox.com/s/tob7fxj9yo33507/xverify.zip?dl=0 and will go to public once the paper is accepted. We did not upload it to Code Ocean because there are dependency libraries which are not supported.).
- We evaluate the effectiveness and efficiency of *XVerify* in comparison with two state-of-the-art tools, VeriSmart [16] and Solicitous [15].

The rest of the paper is organized as follows. In Section 2, we illustrate how *XVerify* works through two simple examples. In Section 3, we present the details of our approach. In Section 4, we discuss the evaluation results. In Section 5, we review related works and lastly we conclude with a discussion on future work in Section 6.

2 Motivating Examples

Given a smart contract, the goal of *XVerify* is to verify the correctness of the properties in form of assertions in the presence of an unbounded arbitrary sequence of function calls. In this section, we demonstrate the key features of *XVerify* through two examples which are adapted from real-world contracts.

Example 1. Figure 2 is a simplified version of the original contract[1], which is reported to CVE (Common Vulnerabilities and Exposures system) for having a "tradeTrap" issue [21]. In this example, users can buy the GRX token through function *buy*, the amount is the multiplication of $msg.value$ and $buyPrice$. The reporter claims that this

1. Contract address: 0x219218f117DC9348b358b8471c55A073E5e0dA0b

multiplication can cause an overflow, which could lead to a financial loss to the buyers, aka the "tradeTrap" issue. However, we will verify this contract with *XVerify* to show that it is a false report.

To verify this contract, *XVerify* first constructs the CFG as shown in Figure 3 based on the EVM bytecode. In this CFG, node $n0$ stands for the state after deploying the contract to Blockchain. It is decided by the constructor function in the contract which is only executed once. There is no explicit constructor function in this example, but a similar function statement exists at line 2, which initialises the storage of $buyPrice$ to 760. Thus, after $n0$, the state becomes $buyPrice = 760$. Node $n1$ is the root node which is the entry of the contract after it is deployed. Anyone can call any function in the contract through this single entry. The nodes without children represent the exits of the contract, like $n4$. The node $n3$-a is an arithmetic node which contains arithmetic operations (i.e., `MUL`) as shown by the bytecode snippet ⓐ beside node $n3$-a. *XVerify* generates an overflow assertion $(msg.value * buyPrice \leq 2^{256} - 1)$ for this operation inherently, and forms a virtual fail node as shown by node $n3\_fail$ in the figure. A virtual fail node does not exist physically, it is only used to facilitate the analysis. Nodes $n4$ and $n6$ are fail nodes, which contain the instruction `ASSERTFAIL` (0xfe). Note that node $n4$ is generated by the compiler, which checks, together with node $n3$-b, whether the divisor $buyPrice$ is 0 in the division $(amount/buyPrice)$ at line 5 in the contract. In snippet b corresponding to node $n3$-b, the execution will jump to node $n5$ if buyPrice is not 0, otherwise, it will go to node $n4$. Node $n6$ stands for the assertion provided by the developers at line 5, which asserts the equality between $msg.value$ and $amount/buyPrice$. Nodes $n7$ and $n8$ are terminal nodes. The dot lines from the terminal nodes to the root node $n1$ form a global loop-like structure due to the fact that all public functions can be called repeatedly. This CFG also demonstrates the discordance on assertions between the source code and the EVM bytecode.

Next, we try to verify all the assertions in the function level by labelling the CFG nodes with an initial contract invariant $true$ at node $n1$. Furthermore, we only concentrate on the nodes which are necessary for the verification of assertions. This is inspired by the idea of lazy annotation, which only deduces the annotation in response to search failures [18]. Thus, we only traverse the nodes which lead to the fail nodes to form the invariants. Take the invariant of node $n3\_fail$ as an example, the part $(msg.value < 2^{100})$ is introduced by *XVerify* for accurate analysis. $msg.value$ stands for the amount of Ether transferred to this contract when the function buy is invoked. We constrain it to be smaller than 2^{100} as the total supply of Ether is around 2^{80} to date. Nobody can deal more Ether than that number. With the node invariant of $(msg.value < 2^{100} \wedge msg.value * buyPrice > 2^{256} - 1)$ for node $n3\_fail$, we will find the virtual fail node is reachable, which means the overflow assertion is possibly violated. The reason is obvious since the number of $buyPrice$ is unknown.

Then, we invoke the contract invariant learning module to conduct the contract level verification. The learning module takes the global variables (e.g., $buyPrice$ in GRX) as features to generate data samples. As we are trying

```
1 contract ContribToken {
2     mapping (address => uint) balances;
3     constructor() public { balances[msg.sender] = 10**4; }
4     function transfer(address receiver, uint numTokens)
          public {
5         require(numTokens <= balances[msg.sender]);
6         balances[msg.sender] = balances[msg.sender]-
              numTokens;
7         balances[receiver] = balances[receiver]+numTokens;
8         assert (balances[receiver] >= numTokens);
9     }
10 }
```

Fig. 4. Example contract adapted from ContribToken

to find a contract invariant, only global variables (a.k.a. storage variables) matter. We execute random sequences of functions in the contract with the randomly generated global variables' valuation and label the valuations according to the rules illustrated in Section 3.3.1. In this example, *XVerify* first samples the valuation of $buyPrice$ as 760. Note that *XVerify* is designed to generate a data sample which satisfies the precondition while starting sampling. The precondition in this example is $(buyPrice = 760)$. We execute the function buy with this valuation and give random input valuations to other variables (e.g., $msg.value$) at the same time. We label this valuation as positive as all the assertions hold (our sampling valuation does not beyond 300), which forms the first data sample $\{(760, +1)\}$. We also sample $buyPrice$ with other valuations like 100. But these data samples are thrown away as they do not satisfy the precondition although they meet the post assertion. Thus, a candidate invariant $true$ is returned because there is only one positive data sample in the sampling phase. A counterexample is generated when we validate this candidate invariant. The 256-bit counterexample is $0X01f1...1$, which is labelled as negative. With this updated dataset $\{(760, +1), (87817...1, -1)\}$, a new candidate invariant is returned, which is $(buyPrice \leq 760)$. Later, this invariant is refined to $(buyPrice > 0 \wedge buyPrice \leq 760)$, and it is successfully validated this time. That means, the arithmetic multiplication is safe, and all the assertions in the contract are also verified simultaneously as long as a valid contract invariant is returned. Detail explanation will be seen in Section 3.

In contrast, the exhaustive verifier VeriSmart [16] emits a warning for this operation and same for Solicitous [15], which is a false alarm actually.

Example 2. The ContribToken contract shown in Figure 4 is adapted from a real-world contract[2]. This contract issues certain amount (10^6) of tokens to the creator by assigning them in the global mapping variable $balances$ in the constructor at line 3. These tokens are then distributed to other users by function $transfer$, which sends $numTokens$ from the message sender's account to the recipient's account (lines 6–7) if the sender's balance is sufficient (line 5). Conventional tools report a false alarm for the assertion at line 8, while *XVerify* verifies it correctly.

To start the analysis, we first try to verify the assertion without the help of the contract invariant after constructing the labelled CFG. Take the `ADD` operation at

2. Contract address: 0x966daed1348fbd894bb6c404d9cddf78a9932913

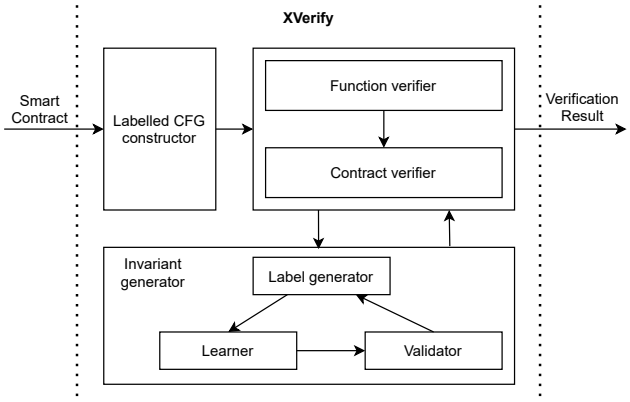

Fig. 5. Flow chart for *XVerify*

line 7 as an example, the node invariant for the virtual fail node is $(balances[msg.sender] >= numTokens) \wedge (balances[receiver] + numTokens < balances[receiver])$. Obviously, it is reachable as the value for $balances[receiver]$ is unconstrained.

Thus, the contract invariant learning is invoked. In this example, a new feature *sum* for mapping is introduced by *XVerify*. This is a useful feature for capturing contract invariants, such as an invariant which states that the sum remains constant or the sum never overflows. For each numeric feature x, we have $(x < uint.max)$ as a standard invariant candidate. At the same time, the new feature *sum* has the following two properties:

$$sum\_balances = \sum_i balances[i] \quad (1)$$

$$summing\ any\ balances[i]\ does\ not\ overflow. \quad (2)$$

The first property ensures the introduced feature *sum* is the total of all accounts in the $balances$ mapping. The second property ensures the summation of any two or more accounts should be smaller than sum. These properties always hold as long as *sum* is involved in and no other functions increase or decrease the total value of mapping variable $balances$.

We then sample valuations for this new feature *sum* and label it by executing all functions in the contracts. As the upper bound for variable valuations is much smaller than 2^{256} while sampling, the assertions for overflow are hardly violated. Thus, only positive data samples are collected and a candidate invariant ($true$) is generated initially. This candidate invariant is later refined to be $sum\_balances \leq 10^4$ with the help of the inherent properties of *sum* illustrated in Eq. (1) and Eq. (2), which is validated to be the real contract invariant candidate. With this contract invariant, all the fail nodes are proved to be unreachable, and we claim the assertions hold in this contract.

VeriSmart reports a timeout on this contract and Solicitous finishes the analysis with multiple false alarms.

3 Our Approach

In this section, we first demonstrate the workflow of *XVerify*. As shown in Figure 5, *XVerify* consists of three parts, i.e., the labelled CFG constructor, the verifier and the invariant generator. It takes as input the EVM bytecode and constructs a labelled CFG at the first step. The constructor will mark the arithmetic nodes and fail nodes, and label the nodes with the respective node invariants. This labelled CFG is fed into the verifier later on. There are two components at this stage, function verifier and contract verifier. Function verifier works without taking the contract invariant into consideration and aims to verify the assertions at the function level. If we fail to verify any of the assertions, that means this assertion can be violated at function level, and thus the contract verifier is invoked. At this step, the contract verifier invokes the invariant generator to systematically infer a sound contract invariant for the assertions. If an invariant is returned by the generator, the verification succeeds. Otherwise, a counter example demonstrating the potential vulnerability is returned to the users.

3.1 Labelled CFG Construction

A CFG is the graphical representation of control flow paths of programs. Given the bytecode of a smart contract, it is constructed based on the assembled EVM code. We omit the definitions for the EVM instruction set, readers can refer to Ethereum yellow paper [1] for further details. A smart contract is composed of a sequence of instructions. Typically the instructions are organized into basic blocks, i.e., a linear sequence of instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed) [22]. Labelled CFG is a CFG labelled with node invariants, which is defined as follows:

Definition 1. Given a smart contract (SC), its labelled CFG is a 4-element tuple $(N, root, E, \mathcal{I})$ where N is a set of nodes representing basic blocks of instructions and these instructions have the same node label; $root \in N$ is the entry node; $E \subseteq N \times N$ is a set of edges; $\mathcal{I} : N \to Pred$ is a mapping that associates each node with a corresponding invariant. $\mathcal{A} : N \to Pred$ is a mapping that associates fail nodes with assertion predicates.

Constructing the CFG in practice is non-trivial. That is, given the EVM bytecode of a smart contract, we first disassemble the bytecode into a sequence of EVM instructions. To identify the edges of the CFG, we must figure out all the targets of `JUMP` and `JUMPI` instructions. These targets are not always available statically. Thus, we dynamically simulate the execution by concentrating on the control flow only. That means we simulate the stack to precisely identify the target of JUMP and JUMPI (since all the control-flow targets are kept in the stack as the EVM is a stack machine). Readers are referred to [23], [24] for details on how the CFG is constructed. We claim the constructed CFG is complete as long as the nodes without children are terminal nodes or fail nodes, whose last instruction is `RETURN`, `REVERT` or `ASSERTFAIL`. This claim is based on the fact that all the addresses of children nodes in EVM bytecode contracts are certain and they are integer numbers counting from number 1[3].

3. https://github.com/ethereum/solidity/blob/develop/libevmasm/Assembly.h\#L52 accessed on Aug.27, 2021

At the same time, we mark the nodes containing opcode ASSERTFAIL (0xfe) as fail nodes. Furthermore, we mark the nodes containing opcodes of ADD, SUB, MUL as arithmetic nodes (DIV is taken into consideration by Solidity compiler as explained in Section 2), and insert a conditional check for overflow after those opcodes for the arithmetic nodes. If the condition is satisfied (i.e., overflow occurs), a fail node is reached; otherwise, the control flow continues with the normal flow. If all the fail nodes are unreachable, the contract is safe.

3.2 Invariant Generation

We introduce the conception of node invariants based on the symbolic semantics and further explain how the invariants are formed at this subsection.

Definition 2 (Symbolic Semantics). Let $(N, root, E, \mathcal{I})$ be a smart contract, its (symbolic) semantics is defined as a transition system $(S, init, \rightarrow_s, O)$, where S is a set of symbolic states, and each state s is a pair (pc, ϕ) where: $pc \in N$ denotes the current program counter, ϕ is a constraint which consists of the symbolic path constraint and the symbolic values of the program variables; $init \in S$ is the initial state and each program variable is initialized to a fresh input symbolic variable while introduced; $\rightarrow_s \subseteq S \times C \times S$ is the transition relation conforming to the symbolic semantic rules (shown in Figure 6), $O \in S$ is the set of final states.

Note that the symbolic values of the program variables are also taken into consideration for the constraint ϕ in each state s which is different from the conventional definition for symbolic state. This is necessary for the verification of intrinsically assertions inserted by us, they can only be inferred with this symbolic values. The set C is the symbolic commands (opcodes) which are based on the EVM instruction set. We show a few most relevant symbolic execution rules of the opcodes in Figure 6 and refer the readers to [1] for the full set of rules. These rules are either particular to EVM or critical for our analysis. Note that, Fr in Figure 6 is a function returning a free symbolic variable for any different input. ϕ is updated by $\phi[op \mapsto Fr(op)]$, which means a new symbolic term is introduced if this opcode is not introduced before, e.g., SHA3(addr) is encoded to be sha3_addr by Fr. The variable $upperbound$ is explained in the NBC-ops in the following.

NBC-ops stands for Numeral Bound Constraint opcodes, which include the opcodes in Table 1. This is where the domain-specific constraints come into picture. We roughly give an upper bound for these attributes. This upper bound is 2^{100} in our implementation. It is a reasonable bound for our analysis as the total issuance of Ether is around 2^{80}. Thus, the value for commands like BALANCE, CALLVALUE, are impossible to exceed the upper bound. FC-ops are opcodes those are Free of Constraints. It is worth mentioning that SHA3 and CALL-like opcodes also drop in this category. We treat SHA3 as an uninterpreted function, which holds the property that the results of same arguments are the same. We assume that any result is possible by CALL-like opcodes. SLOAD and MLOAD are similar, different symbols are used to represent storage and memory. The execution rule for JUMPI is split into two, i.e., JUMPI-1 and JUMPI-2.

TABLE 1
Rules for opcodes

Rule	Opcodes
NBC-ops	BALANCE, CALLVALUE, CALLDATASIZE, GASPRICE, EXTCODESIZE, RETURNDATASIZE, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, SELFBALANCE, MSIZE, GAS
FC-ops	SHA3, ADDRESS, ORIGIN, CALLER, CALLDATALOAD, EXTCODEHASH, BLOCKHASH, COINBASE, CREATE, CREATE2, STATICCALL, CALL, CALLCODE, DELEGATECALL, SLOAD, MLOAD, CALLDATACOPY, EXTCODECOPY, RETURNDATACOPY
JUMPI-1/JUMPI-2	JUMPI
Simp-ops	SUB, DIV, SDIV, MOD, SMOD, EXP, SIGNEXTEND, LT, GT, SLT, SGT, EQ, AND, OR, XOR, BYTE, SHL, SHR, SAR, NOT, ISZERO, SELFDESTRUCT, REVERT, RETURN, STOP, POP, ADDMOD, MULMOD, MSTORE, MSTORE8, SSTORE, DUP1, ..., DUP16, SWAP1, ..., SWAP16, JUMP

JUMPI-1 updates ϕ to be ($\phi \wedge \neg cond$) and updates pc to $pc+1$. JUMPI-2 updates ϕ to be ($\phi \wedge cond$) and pc to T. T is the destination instruction's pc if the $cond$ is satisfied. We omit the rules for Simp-ops in Figure 6. Simp-ops encode their operands into 256 bit-vector symbolic values directly as long as there is a symbolic value in the operands. Otherwise, they are concretely executed.

With the symbolic execution rules, we can form the symbolic trace. A (symbolic) trace tr is a sequence of symbolic states in the form of $tr = \langle s_0, s_1, \ldots, s_{k+1} \rangle$, where $s_0 = init$ and $s_i \rightarrow_s s_{i+1}$ for all $0 \leq i \leq k$. We write $last(tr)$ to denote the last state of the trace, i.e., $last(tr) = s_{k+1}$. The set of symbolic traces, written as $Trace(SC)$, is the set of all traces which can be generated according to the symbolic semantics.

Definition 3 (Node Invariant). Given a smart contract $SC = (N, root, E, \mathcal{I})$, a predicate ψ is an invariant at node n, denoted as $\mathcal{I}(n) = \psi$, if and only if $last(tr) \models \psi$ for all $tr \in Trace(SC)$ s.t. $\pi(last(tr)) = n$.

Intuitively, the above definition of ψ is an invariant at node n if and only if ψ is satisfied by all the traces leading to node n, i.e., when the trace reaches n, its variable valuation satisfies ψ. Function π maps the state to the corresponding node n.

By construction, node invariants are the conjunction of all constraints from path constraints and symbolic valuation constraints. As shown in Algorithm 1, we infer the invariants of node n according to its parent quantity. In the algorithm, if the $parent(n)$ is 0 (lines 10-12), that means this node is the root node, $true$ is returned when verifying in the function verifier, however, it is the contract invariant *XVerify* learned in the contract verifier, e.g., ($s\_n0\_0 \leq 760$) for contract in Figure 2. If the node has more than one parent (line 4-10), we disjunct all the constraints of its parent nodes. Intuitively, this is because n can only be reached via one of its parents. The other parts of the algorithm are quite self-explanatory.

$$\text{NBC-ops}\frac{\phi' = \phi \wedge (Fr(op) < upperbound)]}{(pc, \phi) \longrightarrow_s (pc+1, \phi')} \qquad \text{FC-ops}\frac{\phi' = \phi[op \mapsto Fr(op)]}{(pc, \phi) \longrightarrow_s (pc+1, \phi')}$$

$$\text{JUMPI (cond, T)-1}\frac{\phi' = \phi \wedge \neg cond}{(pc, \phi) \longrightarrow_s (pc+1, \phi')} \qquad \text{JUMPI (cond, T)-2}\frac{\phi' = \phi \wedge cond}{(pc, \phi) \longrightarrow_s (T, \phi')}$$

Fig. 6. Symbolic execution rules

Algorithm 1: Inv Infer $inferI(CFG, n, \phi(root))$

```
1  if |parent(n)| = 1 then
2  |  n' ← parent(n) ;
3  |  return inferI(CFG, n') ∧ φ(n) ;
4  else if |parent(n)| > 1 then
5  |  ζ ← false ;
6  |  for n' ∈ parent(n) do
7  |  |  ζ ← ζ ∨ (inferI(CFG, n') ∧ φ(n)) ;
8  |  end
9  |  return ζ ;
10 else
11 |  return φ(root) ;
12 end
```

Algorithm 2: Lazy Check $lazyCheck(CFG)$

```
1  result ← "unsat";
2  for n ∈ (fail nodes) do
3  |  I(n) ← inferI(CFG, n);
4  |  result = check(I(n));
5  |  if result is "sat" then
6  |  |  return result;
7  |  end
8  end
9  return result;
```

Proposition 1. If all the fail nodes are unreachable, the contract is safe.

With Algorithm 1, we further introduce Algorithm 2 to check the reachability of the fail nodes. If the result is "sat" which means the fail node is reachable, we will terminate the loop and return the result. Otherwise, we will check all the invariants of the fail nodes whic are derived from the assertions in the contract. If the result is "unsat" till the end of the loop, that means all the fail nodes are unreachable, we claim the contract is verified.

3.3 Invariant Learning

In this subsection, we present our "guess and check" approach for loop invariant learning in function verifier first and then extend it to the contract invariant learning in contract verifier regarding to the domain-specific features of smart contracts.

3.3.1 Loop Invariant Learning in Function Verifier

Intuitively, the invariant generator consists of three parts, i.e., *Label generator*, *Learner*, and *Validator*. The details are shown in Algorithm 3 where n is the head node of a loop (i.e., a node representing the start of a loop). In this algorithm, 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., 10. In general, a reasonably large set of random samples are often helpful in learning candidate invariants efficiently. Labelling at line 2 is based on the CFG and the concrete input (a valuation from DS), the program is executed from node n until termination. During the execution, node n may be visited iteratively and the variable valuations upon reaching n are also added to DS. The valuations in DS are categorized into three categories, i.e., '+' for positive, '-' for negative, and 'e' for error. A valuation a which starts from an initial valuation a_0 and becomes a after zero or more iterations is labelled based on whether a_0 satisfies $\mathcal{I}(n)$ and whether eventually an assertion is violated. Specifically, it is labelled

- '+': if a_0 satisfies $\mathcal{I}(n)$, and none of the fail nodes is reached during the execution.
- '-': if a_0 violates $\mathcal{I}(n)$ and any fail node is reached during the execution.
- 'e': if a_0 satisfies $\mathcal{I}(n)$, and any fail node is reached during the execution.

Algorithm 3: Loop Invariant $generateLI(CFG, n)$

```
1  DS = rand(Var);
2  LDS = label(DS, CFG, n);
3  while not timeout do
4  |  (flag, ds) ← checkErr(LDS);
5  |  if !flag then
6  |  |  return ("falsified", ds);
7  |  end
8  |  φ ← learnINV(LDS);
9  |  CE ← validate(φ, CFG, n);
10 |  if CE = ∅ then
11 |  |  return ("succeed", φ);
12 |  else
13 |  |  LDS ← label(CE, CFG, n) ;
14 |  end
15 end
16 return "timeout";
```

The invariant is learned and strengthened from line 3 to line 15. If there is any 'e' valuation in LDS, "falsified" is returned at lines 4-7. A candidate invariant is expected from function $learnINV$ at line 8. The 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 [20], 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.

The learned candidate invariant is then validated by the *Validator*. The *Validator* is constructed according to Hoare logic [25], in which the precondition is the predicates from the node before the loop head node. The postconditions are all the assertions which the traces arrive through the loop head node. The inductiveness is achieved by checking the invariant at the loop head node. A candidate invariant ϕ is indeed an invariant if and only if it is checked to satisfy all the above Hoare rules by the *Validator*. Then, "succeed" is returned together with the validated invariant. Otherwise, the labelled counterexample is added into the data sample set LDS to learn a new candidate by the *Learner* at line 13. The invariant generator will eventually return "timeout" at line 16. Interested readers can refer to [26] for further details.

3.3.2 Contract Invariant Learning in Contract Verifier

Since functions in the contract can be called repeatedly through transactions, we have a "loop" at the contract level, which allows us to extend the "guess and check" approach to the contract invariant learning. This contract-level "loop" is different from ordinary loops in two ways. We explain how to adapt the approach to such a loop regarding the differences. First, there is a constructor function in the contract, which can only be called once. Thus it is excluded from our loops but only provides the precondition in our implementation. Second, there is no terminating conditions (for simplicity, we ignore the instructions such as `SELFDESTRUCT`, which terminates the contract completely in this part of the discussion) in this loop, as the functions can be invoked as many times as possible. Note that this makes it challenging to correctly label the sample. Existing approaches on learning loop invariants run a test until completion and then label it positive only if there is no assertion violation. In our case, this becomes impossible since there is always the possibility that a future function call will result in an assertion failure. For example, if a data sample satisfies the precondition and violates the assertion only after 200 function calls of the same function, it is a counterexample. However, *XVerify* only executes the "loop" 20 times while the assertion still holds. Thus, it is labelled as positive '+'. We remark that such a problem does not mitigate the correctness of our approach nor the effectiveness of our approach on verifying smart contracts. The reason is that such mislabeling only happens if the sample is a true counterexample, in such case we will not be able to verify the smart contract any way.

To start the learning, we randomly generate data samples. Similar to the invariant learning in loops, we generate one set of data sample that satisfies the precondition. Then, we run all the public functions one by one with the same data sample to generate the $iter1$ (iterate 1) states. This process is to simulate the transactions initiated by many users who can call any function with the same input. Based on these $iter1$ states, we further run all the functions again to get the $iter2$ states and further again till $iter5$, which means we only mimic the transactions to call a function 5 times at most. To minimize the computation cost, we only choose the unique states at each iteration to keep on.

Algorithm 4: Overall Verification Algorithm

```
1  CFG ← CFG_construct(SC);
2  result ← lazyCheck(CFG);
3  if result is "unsat" then
4  |  return "succeed";
5  else
6  |  result ← generateLI(CFG, n);
7  |  if result is "succeed" then
8  |  |  return "succeed";
9  |  else
10 |  |  return "warning";
11 |  end
12 end
```

Our observations show that this step is necessary as many functions lead to the same state in practice. With the same rules explained in Section 3.3.1 for labelling, we get the labelled data from the *Label generator* lastly.

The *Learner* and the *Validator* work similarly as before except that the $root$ node acts as the "loop head" now. As long as a candidate invariant is validated by the *Validator*, a true invariant is found and all the assertions in the contract are verified simultaneously.

Example 1. As the contract shown in Figure 2, a candidate invariant $(buyPrice > 0 \wedge buyPrice \leq 760)$ is returned by the Learner. The Hoare rules are as follows: ① The precondition is $buyPrice = 760$; ② Two postcoditions in this example are $buyPrice \neq 0$ and $msg.value = amount/buyPrice$; ③ The inductiveness is checked at the $root$ node. The invariant is the same all the time, since the variable $buyPrice$ is not modified by any function in this contract. Thus, it is trivial to conclude that the Hoare rules are all satisfied, and it is a valid invariant.

3.4 Overall Algorithm

The overall approach is shown in Algorithm 4. Given a smart contract SC, we first construct a labelled CFG at line 1. Then, we try to verify whether each fail node is reachable by $lazyCheck$ at line 2. If the $result$ is "unsat" which means all the fail nodes are unreachable, "succeed" is returned at line 4. Otherwise, there is at least one fail node reached in the analysis. Algorithm $generateLI$ is invoked to invoke the contract invariant in at line 6. If the $result$ is "succeed", a valid contract invariant is generated and all the assertions are verified against this contract invariant. Thus, "succeed" is returned at line 8. If the $result$ is "timeout" or "falsified", a "warning" is returned to the user for manual inspection at line 10.

Theorem 1. Algorithm 4 returns "succeed" if and only if assertion violation is not possible.

Proof: Assume there is an assertion violation detected by our overall verification algorithm, that means there is a trace $tr = \langle s_0, s_1, \ldots, s_n \rangle$ whose last state $s_n \models \psi$, so the corresponding fail node can be reached. If it is detected by the subroutine $lazyCheck$ with the default contract invariant $true$, the $result$ returned must be "sat" and the loop

invariant generation algorithm is invoked. In this algorithm, we return either a concrete false example of the trace or a "timeout" result because the validation of the candidate invariant always fails. Thus, it is impossible for Algorithm 4 to return "succeed" while some assertion is violated. □

4 Implementation and Evaluation

To evaluate the proposed approach in this paper, we have implemented a prototype tool named *XVerify*. It consists of around 6,000 lines of C++ code and the implementation is publicly available at GitHub[4]. *XVerify* takes the smart contracts as input and outputs the verification results. Once a smart contract is given, it will be compiled into EVM bytecode and further disassembled into EVM instructions by Solidity compiler and Ethereum toolkit. With the EVM instructions, *XVerify* starts to construct the labelled CFG and initializes invariants and assertions for each node. Later, LINEARARBITRARY [20], a learning algorithm built on LIBSVM [27] and C5.0 [28] which is capable of learning an arbitrary combination of linear classifiers, is employed to generate loop invariants and contract invariants. Finally, we adopt Z3 solver [29] to check the validity of the generated invariants and specified assertions through bit-vector constraint solving.

4.1 Evaluation

In the following, we evaluate the effectiveness and efficiency of *XVerify* in practice by studying the following research questions (RQ).

- **RQ1:** How effective is *XVerify* in verifying the smart contracts compared to state-of-the-art tools, i.e., Solicitous [15] and VeriSmart [16]?
- **RQ2:** How efficient is *XVerify* in verifying the smart contracts compared to the above two tools?
- **RQ3:** How does *XVerify* perform in detecting the smart contract vulnerabilities when verification fails?

All experiments are conducted on a machine with an Intel Core i7-7700HQ CPU with 8 cores clocked at 2.8GHz, and 23.4GB of RAM, running the system of 64-bit Ubuntu 16.04LTS. The dependancies of *XVerify* include Z3 (version 4.8.8) and the boost library (version 1.68.0). As of now, it is developed for Solidity before version 0.6.10 and EVM toolkit before version 1.9.15.

4.1.1 Benchmarks

We choose Solicitous [15] and VeriSmart [16] for baseline comparison as they are state-of-the-art. In particular, Solicitous is developed with the Ethereum Foundation and is built-in with Ethereum framework. The experiments are conducted with 87 unique Solidity contracts with more than 12k assertions, including assertions from assert statements by users and arithmetic operations automatically generated by *XVerify*. These contracts consist of 57 contracts of version v0.5 and 30 contracts of version v0.6, as Solicitous supports overflow checks since Solidity version v0.6 and VeriSmart only supports v0.5 Solidity contracts. They are all selected from the test subjects reportedly analyzed by Solicitous in [15], and are available through the Etherscan explorer[5]. These contracts are picked out according to the following criteria: ① the top 30 contracts have the most transactions, or ② the top 30 contracts hold the most valuables. We believe these contracts deserve more attentions as they either have wider effects on more users or cause more losses if they are vulnerable. With these standards, 57 v0.5 contracts, three contracts are overlapped, are selected. The contract with the most transactions among them are $1,177,328$ and the one with the maximum balance is $1,271,260$ Ether. 30 v0.6 contracts that have most transactions (1,519) are selected. We omit the top 30 most valuable contracts of v0.6 as almost all the balances of the contracts are 0. Our study shows that the average lines of code for these contracts is 912 and the biggest contract file[6] has around 5,700 lines of code. They are complicated real-world contracts and not easy to verify.

4.1.2 Results

To conduct the experiment, we further acquired the detailed information from Etherscan, such as compiler versions, optimize options and contract names deployed, which are all used for precise bytecode generation. The global wall time limit for all tools are 3,600 seconds and Z3 solver requests for *XVerify* and VeriSmart are 10 seconds (the default wall time is adopted for Solicitous as there is no option for users).

The results are demonstrated in Table 2 and Table 3. In these tables, columns safe, unsafe and unk. (simplified for "unknown", which is due to either exception or timeout) are the results returned by the tools. Column "unsafe" includes two scenarios, i.e., alarm and warning. Scenario *alarm* only effects on *XVerify*. It means, at least, one concrete counterexample that fails an assertion in *Invariant generator* is returned while learning the invariant of a contract. Scenario *warning* for Solicitous is consistent with the raw output of the tool. It is the *unverified* items for VeriSmart, and for *XVerify* it means there is no contract invariant generated to fully verify that contract. The possible reason may be due to the failure of the invariant learning which is caused either by the limited capabilities of our learning module that cannot converge in time or the potential violation of any assertion. Thus, a counterexample which fails the assertion at the previous step, the function level, is returned. To conclude, we are more confident about the vulnerabilities of a contract if it is labelled as *alarm* rather than labelled as *warning* and the vulnerabilities labelled as *alarm* take priority over those labelled as *warning* for users. Columns "TN", "FN", "TP" and "FP", stand for true negatives, false negatives, true positives and false positives, are organized after we manually examined the results between the contract codes and the tools' output. A true negative occurs when tools return "safe" and the contract is indeed "safe" according to our manual check. While a false negative occurs when tools return "safe" but the contract is actually "unsafe". Vice versa, a true positive is the case when tools return "unsafe"

4. The code and the experiment results can be found at https://www.dropbox.com/s/tob7fxj9yo33507/xverify.zip?dl=0. We did not upload it to Code Ocean because there are dependency libraries which are not supported

5. https://etherscan.io as of May. 4th, 2020.

6. Contract address: 0x1e0447b19bb6ecfdae1e4ae1694b0c3659614e4e

```
1 function add(uint a, uint b)internal pure returns(uint c){
2     c = a + b;
3     require(c >= a);
4 }
```

Fig. 7. Require pattern for safeMath

TABLE 2
Comparison Results with Solicitous on v0.6 Smart Contracts

	safe	unsafe		unk.	TN	FN	TP	FP
		alarm	warning					
Solicitous	0	N.A.	28	2	N.A.	N.A.	N.A.	N.A.
XVerify	7	7	13	3	7	0	6	14

and the contract is "unsafe" and a false positive occurs when tools return "unsafe" but the contract is actually "safe".

In the process of manually checking the results by VeriSmart, we found multiple false positives which are caused by the same pattern shown in Figure 7. It is quite clear that, if there is an overflow in the `ADD` operation, the transaction will be reverted by line 3. However, VeriSmart fails to catch such guard conditions. In contrast, we avoid such false positives. *XVerify* attaches the operands a and b to the result c, and records the potential overflow constraint in the meantime at line 2. Then, when executing to line 3, the result c is involved in a comparative operation, and the other operand is one of the operands attached to the result c, moreover, the following node is a revert node, we conclude that the comparative operation is to avoid addition overflow problems by the developers. Same methods are also applied to other arithmetic operations like `MUL`, `SUB` etc.

To make the comparison fair, three sets of data by VeriSmart are shown in Table 3. Row VeriSmart1 is the result which takes the pattern in Figure 7 as a false positive. Row VeriSmart2 and VeriSmart3 are both the results which ignore the effect of the pattern in Figure 7. We demonstrate them both here is because VeriSmart returns all the analysis results of the arithmetic operations and assertions for a contract. Some of these results are correct, some are not. The difference between row VeriSmart2 and VeriSmart3 lies only in "TP" and "FP" columns. A contract is categorized into "FP" if there are still other false positives except above "require" pattern in VeriSmart2. A contract is categorized into "TP" instead of "FP" as long as a true positive is returned except above "require" pattern in VeriSmart3. For example, if there is only one warning caused by the "require" pattern in Figure 7 for a contract, this contract is categorized into "warning" and viewed as "FP" in row VeriSmart1, but it is categorized into "safe" and viewed as "TN" in row VeriSmart2 and VeriSmart3. It is the reason that the number of columns "safe" and "TN" is bigger than that in row VeriSmart2 and VeriSmart3.

4.1.3 *Effectiveness of* XVerify

We illustrate the effectiveness of *XVerify* with the number of correctly verified contracts (TN) shown in Table 2 and Table 3.

In Table 2, *XVerify* reports 7 contracts as "safe" and these contracts are true negatives after a carefully manual check. They are successfully verified. In contrast, this number for Solicitous is 0. A manual study of the results by Solicitous shows that, it blindly emits warnings for almost all the arithmetic operations in these contracts except for two unknowns. It is quite challenging and discouraging for users to identify true positives from plenty of false positives. Thus, we did not further categorize those results. Back to *XVerify* , there is one contract among the 7 "safe" contracts which is verified with the help of the contract invariant. Same results also manifest in Table 3. In this comparison experiment, *XVerify* returns 13 contracts as "safe" and they are indeed true negatives after our manual examination. 3 out of 13 contracts are verified with the assistance of contract invariants. As a comparison, VeriSmart returns 3 "safe" contracts, shown at row VeriSmart1, if we take the pattern in Figure 7 as false positives. 2 of these 3 contracts are false negatives because VeriSmart fails to catch the assertion statements in the contracts. If we ignore the false positives caused by the "require" pattern, the "safe" contracts will be 8 as shown by VeriSmart2. However, another one more false negative is also brought in besides the previous two described in VeriSmart1. This false negative is also caused by the failure to capture the assertion statement.

TABLE 3
Comparison Results with VeriSmart on v0.5 Smart Contracts

	safe	unsafe		unk.	TN	FN	TP	FP
		alarm	warning					
VeriSmart1	3	N.A.	14	40	1	2	2	12
VeriSmart2	8	N.A.	9	40	5	3	4	5
VeriSmart3	8	N.A.	9	40	5	3	9	0
XVerify	13	8	14	22	13	0	10	12

We also identified two more kinds of vulnerabilities which cannot be detected by other two tools. They are the boundary checks for *array* and *enum* inserted by Solidity compiler.

To answer RQ1: *XVerify* can correctly verify a pleasant number of smart contracts and it is quite competitive, 7 vs 0 and 13 vs 1/5 in true negative (TN), in comparison with Solidity official tool Solicitous and VeriSmart. We can also detect two more vulnerabilities at the same time.

4.1.4 *Efficiency of* XVerify

We illustrate the efficiency of *XVerify* by comparing the average time consumed during the analysis. As shown in Table 4, column "Fin./To." stands for "Finished/Total", which is the number of contracts finished analysis over the total contracts. Column "Time" is the average time consumed for the contracts which finished the analysis by each tool on these test subjects.

In this table, *XVerify* averagely takes about $100.2\,\mathrm{s}$ to finish analyzing 27 (out of 30) contracts in v0.6 test repository. While Solicitous finishes 28 contracts with an average of $174.1\,\mathrm{s}$. For the v0.5 test repository, *XVerify* performs much better with an average time of $95.8\,\mathrm{s}$ over $1583.6\,\mathrm{s}$ and finishes more contracts at the same time. The reason for the advantage may be that *XVerify* terminates analysis once an alarm or warning arises. We believe this setting is reasonable, as we are uncertain about the subsequent

TABLE 4
Average time consumption for verification

v0.6	Fin./To.	Time (s)	v0.5	Fin./To.	Time (s)
Solicitous	28/30	174.1	VeriSmart	17/57	1,583.6
XVerify	27/30	100.2	*XVerify*	35/57	95.8

```
1  function sendBatch(address[] recipients, uint[] values){
2      // require (recipients.length == values.length);
3      for (uint i = 0; i < recipients.length; i++) {
4          _fullTransfer(msg.sender, recipients[i],values[i]);
5      }
6  }
7
8  bytes32[] values;
9  mapping (bytes32 => uint256) indexes;
10 function remove(bytes32 value) public {
11     uint256 valueIndex = indexes[value];
12     if (valueIndex != 0) {
13         uint256 lastIndex = values.length - 1;
14         bytes32 lastvalue = values[lastIndex];
15
16         uint256 toDeleteIndex = valueIndex - 1;
17         values[toDeleteIndex] = lastvalue;
18         ...
19     }
20 }
```

Fig. 8. Alarm cases by *XVerify*

results after the alarm or warning. In contrast, Solicitous and VeriSmart return the results of all arithmetic operations and assertions in Solidity source code to users. This process costs too much time.

To answer RQ2: *XVerify* outperforms Solicitous and VeriSmart on the sacle of time consumption.

4.1.5 *Detecting Vulnerabilities by* XVerify

We study the capabilities of *XVerify* in detecting the contract vulnerabilities when the verification fails.

As shown in Table 2, there are 7 alarms and 13 warnings returned by *XVerify*. In Table 3, there are 8 alarms and 14 warnings. Note that, an "unsafe" contract is always categorized into "FP" as long as there is one incorrect alarm or warning by *XVerify*. *XVerify* stops analyzing once an alarm or warning arises, which is different from Solicitous and VeriSmart. The 6 true positive (TP) contracts consist of 5 from 7 alarms and 1 from 13 warnings in Table 2. Same situation happens in Table 3, too. These 10 TPs are comprised of 7 from 8 alarms and 3 from 14 warnings.

As we have discussed before, we are more confident about the correctness in detecting vulnerabilities by alarms. They are usually generated by a concrete counterexample which satisfies the precondition and violates an assertion while labelling. One example of such an alarm is shown in Figure 8. The function $sendBatch()$ is to transfer values to recipients one by one with a for loop. There are two implicit assertions which are $i < recipients.length$ and $i < values.length$ in the loop. They are generated automatically by Solidity compiler for arrays to ensure the operations are within the boundaries. The alarm raises when the length of values is smaller than that of recipients.

There are also 3 similar FPs from alarms in Table 2 and Table 3 by *XVerify*. As the function $remove()$ shown in Figure 8, an array $values$ records the contents of input value and a mapping $indexes$ records the corresponding index for each value. The implicit assertion for array boundary by Solidity compiler is inserted at line 17, which is $toDeleteIndex < values.length$. When some value is to remove, the function will retrieve the index of that value from the mapping at line 11 and then overwrite it with the last value in $values$ array at line 17. In such contracts, the developers keep the indexes accord with the values all the time. That means, the index $toDeleteIndex$ at line 17 will never beyond the boundary of the array. However, *XVerify* can not infer the relationship between array $values$ and mapping $indexes$, which generates these 3 FPs.

On the other hand, column "warning" brings in most of the FPs, i.e., 12 out of 14 FPs in Table 2 and 11 out of 12 FPs in Table 3. The reasons come from 3 aspects: ① Fail to learn loop invariants. There are some complicate loops in the contracts which beyond *XVerify* 's capability this moment, like nested loops. ② Insufficient time for z3 solver. Some constraints are complicate enough or some constraints involve in the modulo operation, which makes the solving almost impossible. ③ Unsupported types by *XVerify* currently. Some data types are not supported, like nested mappings, i.e., mapping (address $\Rightarrow mapping(address \Rightarrow struct))etc.$

To answer RQ3: *XVerify* can pinpoint the vulnerable contracts if it returns "alarm" to users, but may be not so accurate if "warning" is returned.

4.1.6 *Threats to Validity*

There are several threats in our evaluation. Firstly, we select the top30 contracts in regard of most transactions and most balance as our benchmarks to avoid the representative problem, although the duplicates have been removed. Secondly, different specialities on different Solidity contract versions by the comparison tools render us to split the test repository into v0.6 and v0.5 to avoid unfair problems. Thirdly, we suppress some false warnings introduced deliberately by the compiler with specific patterns. For example, Solidity generates 0xFF(64Fs)... by $(0 - 1)$. Lastly, we categorize the analysis results by tools into TPs, TNs etc. with our manual examination. This work is quite challenging and may be erroneous in some cases although we have tried our best. One last reflection is about the boundary checks for arrays, structs, enum etc. by Solidity compiler, these checks are compiled into instruction "0xfe" which is the same as assertions. We think maybe it is more reasonable to compile them into "0xfd" which corresponds to requirements instead of assertions.

5 Related Work

In this section, we review the works closely related to ours, mainly on the verification of smart contracts and also a brief illustration on the invariant generation techniques.

5.1 Verification on Smart Contracts

Existing verification works on smart contracts can be roughly categorized into two categories, i.e., function-level verification and contract-level verification.

Many early static analysis works focus on detecting vulnerabilities by scanning specific patterns in path conditions and traces, like identifying timestamp dependency

by Oyente [30], detecting suicidal smart contracts by sCompile [24] and prodigal contracts by Maian [31]. Also, there are other similar well-known tools, like Mythril [32], Osiris [33], Manticore [34] etc. These works are more testing than verification and they can hardly provide any soundness or completeness guarantees.

Some other works are based on theorem provers, like Kevm [35] and Ksolidity [36]. They are both based on the K framework to develop executable formal semantics on EVM and Solidity. They can check contracts against given specifications with the help of deductive program verifiers. Sidney ect. [37] extends an existing EVM formalisation in Isabelle/HOL by a sound program logic at the level of bytecode to prove the correctness of properties for smart contracts. These works provide the capabilities to capture precise vulnerabilities in the formal semantics of the contracts. However, the process can be quite cumbersome usually as the properties checked also need to be formalized in the language of the theorem prover, which damages the usability for contract developers.

Also, some works leverage the existing facilities to verify the contracts. Securify [11] symbolically encodes the dependence graph from the EVM bytecode into Datalog and targets specific types of bugs encoded as data patterns to prove if a property holds or not. However, Securify does not support numerical analysis and cannot be used for finding arithmetic bugs. [38] and Zeus [12] translated smart contracts into intermediate representations like F* programs and LLVM bitcode respectively, then leverage existing tools for F* and Seahorn to reason about the contract correctness. VerX [13] introduces delayed predicate abstraction approach based upon symbolic execution to verify temporal safety specifications written in PastLTL by users. However, VerX does not perform abstraction refinement; thus, the counterexamples it produces can be spurious, moreover, the successful verification may require the users to provide additional predicates [39]. A similar work is also done by Smartpulse [39] recently. It differs from VerX in that it is not limited to safety problems, but also capable of checking liveness properties and never produces spurious counterexamples. However, they all require the users to clearly illustrate the properties with their specification language, and they can even achieve full verification if the specification is detailed enough, which may be a challenge for the developers.

In a word, above works focus on the functional safety of smart contracts and lack the ability for contract level reasoning although some tools can mitigate such a situation with detailed specifications. Some recent works have turned to this problem. Celestial [40] is a work which translates the contracts and the specifications to F* to formally verify the contracts. It allows the users to annotate properties of interest with a Solidity-style specifications. Verisol [41] formalizes semantic conformance of smart contracts against a state machine model with access-control policy. It is a highly-automated formal verifier for Solidity which can produce proofs as well as counterexamples. solc-verify [14] translates 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. These works all require users to manually provide specifications or invariants in some forms, which can be thought as semi-automated tools.

Solicitous [15], the formal engine inside the official Solidity compiler, directly models the Solidity contracts with constrained Horn clauses and leverages the generic theorem provers for fully verification. However, maybe limited by the capabilities of CHC solver on invariants, the performance is not so pleasant in the experiment. VeriSmart [16] automatically discovers transaction and loop invariants with the help of domain-specific refinement to verify smart contracts. However, the refinement is limited in certain simple linear forms, non-linear and compound invariants are not taken into consideration.

5.2 Invariant Generation Techniques

Invariant generation is a long standing problem which is important for program analysis and automated verification. Many approaches have been proposed for loop invariant generation, including those based on abstraction interpretation [42], [43], [44], those based on counterexample-guided abstraction refinement [45], [46] or interpolation [47], [48], those based on constraint solving and logical inference [49], [50], [51] and those based on guess-and-check machine learning approaches [19], [20]. We leverage the approach of machine learning loop invariant generation in this work, since the other approaches often suffer from scalability issues.

6 Conclusion

We leverage the static analysis techniques (i.e., lazy annotation and loop invariant generation techniques) to design and implement a formal verifier called *XVerify* for EVM bytecode contracts. We evaluate it on 87 contracts with more than 12k assertions through a comparison experiment with two state-of-the-art tools, VeriSmart [16] and Solicitous [15]. The experiment results show that *XVerify* performs well on effectiveness and efficiency over the other tools. We will further improve the tool to achieve better generality for practical use in the future.

References

[1] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[2] B. Xu, D. Luthra, Z. Cole, and N. Blakely, "Eos: An architectural, performance, and economic analysis," *Retrieved June*, vol. 11, p. 2019, 2018.

[3] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.

[4] L.-W. Wong, G. W.-H. Tan, V.-H. Lee, K.-B. Ooi, and A. Sohal, "Unearthing the determinants of blockchain adoption in supply chain management," *International Journal of Production Research*, vol. 58, no. 7, pp. 2100–2123, 2020.

[5] P. Laurent, T. Chollet, M. Burke, and T. Seers, "The tokenization of assets is disrupting the financial industry. are you ready?" *Inside. Triannual insights from Deloitte*, no. 19, pp. 62–67, 2018.

[6] G. Zyskind, O. Nathan *et al.*, "Decentralizing privacy: Using blockchain to protect personal data," in *2015 IEEE Security and Privacy Workshops*. IEEE, 2015, pp. 180–184.

[7] Solidity. Solidity documentation. [Online]. Available: https://docs.soliditylang.org/en/latest/

[8] M. Kaleem, A. Mavridou, and A. Laszka, "Vyper: A security comparison with solidity based on common vulnerabilities," in *2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE, 2020, pp. 107–111.

[9] S. Palladino. The parity wallet hack explained. [Online]. Available: https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/

[10] P. Technologies. A postmortem on the parity multi-sig library self-destruct. [Online]. Available: https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/

[11] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[12] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." in *NDSS*, 2018.

[13] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE Symposium on Security and Privacy, SP*, 2020, pp. 18–20.

[14] Á. Hajdu and D. Jovanović, "solc-verify: A modular verifier for solidity smart contracts," in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2019, pp. 161–179.

[15] M. Marescotti, R. Otoni, L. Alt, P. Eugster, A. E. Hyvärinen, and N. Sharygina, "Accurate smart contract verification through direct modelling," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2020, pp. 178–194.

[16] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1678–1694.

[17] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[18] K. L. McMillan, "Lazy annotation for program testing and verification," in *Proceedings of the 22Nd International Conference on Computer Aided Verification*, ser. CAV'10, 2010, pp. 104–118.

[19] J. Li, J. Sun, L. Li, Q. L. Le, and S.-W. Lin, "Automatic loop-invariant generation and refinement through selective sampling," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 782–792.

[20] H. Zhu, S. Magill, and S. Jagannathan, "A data-driven chc solver," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 707–721, 2018.

[21] "Cve-2018-11446," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-11446, accessed: 2021-08-27.

[22] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[23] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "Analyzing smart contracts: From evm to a sound control-flow graph," *arXiv preprint arXiv:2004.14437*, 2020.

[24] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang, "sCompile: Critical Path Identification and Analysis for Smart Contracts," in *Proceedings of the 21st International Conference on Formal Engineering Methods, ICFEM 2019*, 2019, pp. 286–304.

[25] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[26] B. Gao, L. Shi, J. Li, J. Chang, J. Sun, and Z. Yang, "sverify: Verifying smart contracts through lazy annotation and learning," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2021, pp. 453–469.

[27] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[28] J. Quinlan, "C5.0: An informal tutorial," http://www.rulequest.com/see5-unix.html, 2017.

[29] L. De Moura and N. Bjørner, "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*, ser. TACAS'08/ETAPS'08. Springer-Verlag, 2008, pp. 337–340.

[30] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[31] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 653–663.

[32] "Mythril: a security analysis tool for evm bytecode," https://github.com/ConsenSys/mythril, accessed: 2021-08-27.

[33] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 664–676.

[34] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[35] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 204–217.

[36] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun, "Semantic understanding of smart contracts: Executable operational semantics of solidity," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1695–1712.

[37] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in isabelle/hol," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2018, pp. 66–77.

[38] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, and N. Kobeissi, "Formal verification of smart contracts: Short paper," in *PLAS*. ACM, 2016, pp. 91–96.

[39] J. Stephens, K. Ferles, B. Mariano, S. Lahiri, and I. Dillig, "Smartpulse: Automated checking of temporal properties in smart contracts," in *IEEE S&P*, 2021.

[40] S. Dharanikota, S. Mukherjee, C. Bhardwaj, A. Rastogi, and A. Lal, "Celestial: A smart contracts verification framework," Microsoft, Tech. Rep. MSR-TR-2020-43, December 2020. [Online]. Available: https://www.microsoft.com/en-us/research/publication/celestial-a-smart-contracts-verification-framework/

[41] Y. Wang, S. K. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, I. Naseer, and K. Ferles, "Formal verification of workflow policies for smart contracts in azure blockchain," in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2019, pp. 87–106.

[42] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1978, pp. 84–96.

[43] A. Miné, "The octagon abstract domain," *Higher-order and symbolic computation*, vol. 19, no. 1, pp. 31–100, 2006.

[44] V. Laviron and F. Logozzo, "Subpolyhedra: A (more) scalable approach to infer linear inequalities," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2009, pp. 229–244.

[45] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with blast," in *International SPIN Workshop on Model Checking of Software*. Springer, 2003, pp. 235–239.

[46] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *Journal of the ACM (JACM)*, vol. 50, no. 5, pp. 752–794, 2003.

[47] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," *ACM SIGPLAN Notices*, vol. 39, no. 1, pp. 232–244, 2004.

[48] K. L. McMillan, "Lazy abstraction with interpolants," in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 123–136.

[49] A. Gupta and A. Rybalchenko, "Invgen: An efficient invariant generator," in *International Conference on Computer Aided Verification*. Springer, 2009, pp. 634–640.

[50] S. Gulwani, S. Srivastava, and R. Venkatesan, "Constraint-based invariant inference over predicate abstraction," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2009, pp. 120–135.

[51] I. Dillig, T. Dillig, B. Li, and K. McMillan, "Inductive invariant generation via abductive inference," *Acm Sigplan Notices*, vol. 48, no. 10, pp. 443–456, 2013.