Published in Formal methods and software engineering: 24th International Conference on Formal Engineering Methods, ICFEM 2022, Brisbane, November 21-24: Proceedings. DOI: 10.1007/978-981-99-7584-6_2

An Idealist's Approach for Smart Contract Correctness

Tai D. Nguyen[1(✉)], Long H. Pham[1], Jun Sun[1], and Quang Loc Le[2]

[1] Singapore Management University, Singapore, Singapore
{dtnguyen.2019,hlpham,junsun}@smu.edu.sg
[2] University College London, London, UK
loc.le@ucl.ac.uk

Abstract. In this work, we experiment an idealistic approach for smart contract correctness verification and enforcement, based on the assumption that developers are either desired or required to provide a correctness specification due to the importance of smart contracts and the fact that they are immutable after deployment. We design a static verification system with a specification language which supports fully compositional verification (with the help of function specifications, contract invariants, loop invariants and call invariants). Our approach has been implemented in a tool named iContract which automatically proves the correctness of a smart contract statically or checks the unverified part of the specification during runtime. Using iContract, we have verified 10 high-profile smart contracts against manually developed detailed specifications, many of which are beyond the capacity of existing verifiers. Specially, we have uncovered two ERC20 violations in the BNB and QNT contracts.

1 Introduction

> *"After this decade, programming could be regarded as a public, mathematics-based activity of restructuring specifications into programs."*
>
> *(Edsger W. Dijkstra, 1969)*

And it didn't happen. Worse yet, the idea of having a formal specification either before or alongside with a program has become unimaginable for ordinary programmers nowadays.

We however may not have the luxury NOT to have a correctness specification when it comes to smart contracts. Smart contracts are programs that run on top of blockchain. They are often used to implement financial applications and increasingly other critical applications. A bug in a smart contract thus could result in a massive loss of valuable digital assets, which has been demonstrated time and time again [8,22]. More importantly, due to the immutability of blockchain (which is one of its fundamental properties), a smart contract cannot be patched once it is deployed. In other words, once deployed, a bug in the smart contract would make it forever vulnerable. We thus must make sure a smart contract is correct before it is deployed.

Existing approaches on tackling the correctness of smart contracts can be roughly categorized into two groups, i.e., those approaches which target common vulnerabilities and those which support (manually specified) full correctness specification. The former includes an extensive list of approaches and tools on static analysis (such as Mythril [18], Oyente [16] and Securify [28]), fuzzing (such as sFuzz [20], Echidna [13], and ConFuzzius [27]), as well as runtime monitoring (such as sGuard [19], Solythesis [15], and Elysium [12]). While the approaches are different, what is common across these approaches is that they all focus on a collection of generic bugs (such as reentrancy, overflow or underflow, frontrunning and frozen funds). While these approaches are undoubtedly useful, they are incapable of identifying contract-specific bugs or showing their absence.

In this work, we propose iContract, a fully compositional verification system for verifying and enforcing the correctness of smart contracts. iContract supports a rich specification language which allows developers to specify not only the traditional loop invariants and function specifications but also contract invariants (for contract-level specification) and call invariants (for specification of external function calls). We remark that designing a specification language that is relatively easy to use (which is essential in practice), expressive, and makes verification easy is nontrivial. For instance, a smart contract often interacts with other contracts via interfaces. Mishandling such interfaces (e.g., assuming that no contract states are modified by such interfaces or contracts states can be modified arbitrarily) would hinder the verification of contracts. In this work, we annotate external function calls with call invariants (so that we can quantify the behavior of the external function call using a correctness logic formula as well as an incorrectness logic formula). These call invariants can be validated at the runtime and relied upon as assumptions when we verify the calling function.

To evaluate the effectiveness and applicability of iContract in practice, we apply iContract to verify 10 real-world high-profile contracts. For each contract, a full specification of its correctness is first developed manually, with a total of 1 PhD-month. iContract is then applied to verify each of the contracts. The results show that iContract not only is scalable for verifying real-world contracts but also uncovering contract-specific bugs. The results are encouraging as it shows that developing a specification for critical but relatively simple programs such as smart contracts is entirely feasible.

To sum up, our main contributions are as follows. First, we propose an approach for the correctness specification of smart contracts which facilitate completely compositional verification, including revert specification (i.e., specifications that capture explicit reverts) as well as call invariants for frame conditions. Second, we develop an implementation of the compositional verification approach for real-world Solidity smart contracts. Lastly, we conduct an evaluation using 10 real-world high-profile smart contracts (with a full specification of their correctness).

2 Overview

2.1 Smart Contracts

The concept of smart contracts was first proposed by Nick Szabo in 1997 [26]. However, it only became a reality after the creation of Ethereum [30] in 2015. An Ethereum smart contract implements a set of rules that aim to manage digital assets in Ethereum accounts including externally owned accounts and contract accounts. Despite a large variety of contract programming languages (e.g., Solidity [5], Vyper [7], and Bamboo [1]), Solidity is the most dominant one for implementing smart contracts. It is a Turing-complete, object-oriented, and statically-typed programming language. A smart contract in Solidity is similar to a class in object-oriented programming languages such as Java or C#. It contains storage variables that stores persistent data and functions. While public functions can be invoked from other accounts to modify storage variables, private functions are internally invoked by other functions. An example of contract written in Solidity is shown in Fig. 1.

2.2 Vulnerability and Correctness

Same as traditional programs, smart contracts can have bugs. For instance, a long list of common bugs have been identified [6], some of which have been exploited and huge financial losses have occurred [8]. Making sure that a smart contract does not repeat the same mistakes merely constitutes the first step towards contract correctness.

An ideal approach for smart contract correctness verification must satisfy the following requirements. First, it must support a rich notion of correctness. This is because each contract is designed for a unique purpose and thus is expected to satisfy a contract-specific specification. Existing approaches that are designed to verify smart contracts against common general vulnerabilities are thus insufficient. Second, it must be fully compositional, i.e., given a contract, we should be able to establish its correctness without relying on external contracts. Furthermore, each functional unit, such as a function or even a loop, should have its own specification so that any kind of global reasoning (even at the contract level) could be avoided. In so doing, the verification system could achieve scalability. Third, it must be fully automatic once the specification is provided. Lastly, it must guarantee that the smart contract satisfies its specification, either through static verification (ideally) or runtime verification (if necessary).

We obviously must pay some price to achieve the above-mentioned goals. Our approach is thus based on two assumptions. First, we make the strong assumption that developers are either requested or required (by stakeholders or certification boards) to provide a correctness specification. While it was sadly proven too strong an assumption for ordinary programs, it may be justifiable for smart contracts due to the reasons mentioned above. Second, we make the assumption that developers are willing to pay some reasonable amount of additional fee (i.e., for runtime checking) in order to guarantee that the smart contract satisfies the specification.

2.3 An Illustrative Example

In the following, we illustrate how our goals are achieved by iContract through an example. Figure 1 shows a token-issuing smart contract (written according to the ERC20 standard [11]), which is a simplified version of a real-world smart contract named HEALTH[1]. The contract includes global variables *burnFee*, *devFee*, *bFee*, *uniswapV2*, and *balances*. It supports (through a public function) *transfer* of HEALTH tokens (hereafter h-tokens) from account *from* (a.k.a. sender) to account *to* (a.k.a. receiver). Note that the sender is charged with some fee for the transfer. Furthermore, in some cases, it burns (subtracts) an amount (proportional to *value*) of the h-tokens hold by *uniswapV2*, which is a service that swaps h-tokens with BNB (i.e., a token which is often used for token exchange services) or vice versa. Particularly, first, at lines 8–12 if the receiver is *uniswapV2*, the contract swaps *numTokensSell* h-tokens for BNB (line 9). Second, at lines 13–19, if the sender is not *uniswapV2*, the contract burns some h-tokens from *uniswapV2* (line 15). Lastly, at lines 20–26, the contract charges development fee (line 25), burns token (line 26), and transfers the remaining (line 24) to receiver.

To verify the contract, we start with developing a correctness specification. For instance, lines 3–4, 10–11 and 18 constitute the correctness specification of the function *_transfer*. The specification relies on a set of pre-defined functions, such as reverts_if(p), modifies($\overline{x}$), ensures(p, q), call_modifies($\overline{x}$) and call_inv(p, q). Intuitively, reverts_if(p) says that the transaction reverts if p is satisfied; modifies($\overline{x}$) (respectively call_modifies($\overline{x}$)) says that the function (respectively the external call) only modifies those variables in $\overline{x}$; ensures(p, q) is equivalent to the Hoare triple $\{p\}s\{q\}$ where s is the function body; and call_inv(p, q) right after a function call is a call invariant, where p is a pre-condition of the call and q is expected to be satisfied after the call. We remark that modifies($\overline{x}$), call_modifies($\overline{x}$) can be regarded as syntactic sugars of certain special cases of ensures(p, q) and call_inv(p, q).

In particular, the specification at line 4 demands that when $value = 0$, no token should be burned. This is important as burning h-tokens reduces the total supply and, thus, increases the price of h-tokens. If h-tokens can be burned unintentionally (e.g., when $value = 0$), attackers could potentially use the function to manipulate the market price. According to the call_modifies($\overline{x}$) at line 11, only variables *_balances*[*this*] and *_balances*[*uniswapV2*] are modified. The call invariants at lines 10–11 state that the function call at line 9 transfers *numTokensSell* h-tokens from address *this* to address *uniswapV2*. In particular, the *balances*[*this*] is reduced and *balances*[*uniswapV2*] is increased by the same amount (i.e., *numTokenSell*). By default, all global variables could be modified in the called function. Line 18 specifies that no variables are modified by the external call.

Once the specification is given, iContract systematically verifies the contract against the specification. It reports that the specification at line 4 is falsified with a counterexample, i.e., if the sender is not *uniswapV2* and *value* is

[1] deployed at BNB chain address 0x32b166e082993af6598a89397e82e123ca44e74e.

```
1  contract Health {
2    ...
3    /// reverts_if(_balances[from] < value)
4    /// ensures(to != uniswapV2 && value == 0 && _balances[from] >= value, _balances
         [_burnAddress] == old(_balances[_burnAddress]))
5    function _transfer(address from, address to, uint value) private {
6      require(_balances[from] >= value);
7      // require(value > 0);
8      if (to == uniswapV2) {
9        UniswapRouter(uniswapV2).swapAndLiquify(numTokensSell);
10       /// call_inv(_balances[this] >= numTokensSell, _balances[this] == old(
             _balances[this]) - numTokensSell && _balances[uniswapV2] == old(
             _balances[uniswapV2]) + numTokensSell)
11       /// call_modifies(_balances[this], _balances[uniswapV2])
12     }
13     if (from != uniswapV2) {
14       uint burnValue = _balances[uniswapV2].mul(burnFee).div(1000);
15       _balances[uniswapV2] = _balances[uniswapV2].sub(burnValue);
16       _balances[_burnAddress] = _balances[_burnAddress].add(burnValue);
17       IPancakePair(uniswapV2).sync();
18       /// call_modifies()
19     }
20     uint devValue = value.mul(devFee).div(1000);
21     uint bValue = value.mul(bFee).div(1000);
22     uint newValue = value.sub(devValue).sub(bValue);
23     _balances[from] = _balances[from].sub(value);
24     _balances[to] = _balances[to].add(newValue);
25     _balances[address(this)] = _balances[address(this)].add(devValue);
26     _balances[_burnAddress] = _balances[_burnAddress].add(bValue);
27   }
28
29   function transfer(address to, uint value) public returns(bool) {
30     _transfer(msg.sender, to, value);
31     return true;
32   }
33 }
```

Fig. 1. A sample contract

0, h-tokens are burned from *uniswapV2* on line 15. In other words, this contract could be exploited by abusing the function *_transfer* to burn h-tokens and manipulate its price, i.e., an attacker first buys some h-tokens, repeatedly calls *_transfer* as described above, and sells his h-tokens at a higher price.

With the verification result, we can prevent the manipulation by adding one statement *require*(*value* > *0*) at line 7. Afterwards, iContract reports that the specification is successfully verified. This is because if *value* = 0, the function is reverted. Furthremore, if the user wish to verify the revert, he could annotate another specification as reverts_if(value=0) and invoke iContract to verify it. Indeed, our system could verify the revert scenario successfully. Alternatively, if the user chooses to conduct runtime verification, iContract automatically translates the above-mentioned unverified specification into an assertion, which is then validated every time the function is invoked. Note that in the latter case, additional gas will be paid (for executing the assertion) for the correctness.

3 Specification Language

3.1 High-Level Overview

In the following, we present our specification language which is designed to support fully compositional verification of smart contracts at the function level. At a high-level, our specification is composed of function specifications, loop invariants, (external) call invariants and contract invariants.

Function Specifications: Ideally, a user would be able to read the function specification and be fully aware of what the function does. Given a function f, a function specification takes the form of multiple ensures(p, q) statements (at the beginning of the function body), where p and q are predicates that we shall define shortly. Each ensures(p, q) statement represents a Hoare triple $\{p\}f(\overline{x})\{q\}$, i.e., any reachable state at the end of the function (i.e., without reverting) from a state satisfying p must satisfy q. In other words, q is an over-approximation of the states reachable from p.

Loop Invariants: It is well known that loops are difficult when they come to program verification. While there are many existing approaches on synthesizing loop invariants [10,17], for now, we make the assumption that loop invariants are provided as a part of the specification. A loop invariant takes the form of multiple loop_inv(q) statements at the beginning of the loop. Given a loop *while* b *do* s, loop_inv(q) at the beginning of the loop represents a Hoare triple $\{b \wedge q\}s\{\neg b \wedge q\}$.

Call Invariants: Smart contracts often rely on other smart contracts through external function calls. To avoid global analysis, we assume that each external call is associated with a specification in the form of multiple call_inv(p, q) statements and multiple achieves(p, q) statements. These help to ensure the function call behaves expectedly, i.e., they serve as the minimal requirements on the external contracts that are needed to guarantee the correctness of this contract. Given a function call $m(\overline{e})$, a statement call_inv(p, q) forms a triple $\{p\}m(\overline{e})\{q\}$. If p is satisfied before the call, q is always satisfied after the execution of the function call. Such statements can be used to prevent the well-known reentrancy vulnerability. A statement achieves(p, q) forms a specification in the incorrectness logic [21], which intuitively means that if p is satisfied, it is possible to satisfy q by making the external call.

Contract Invariants: A contract invariant takes the form of multiple cinv(p) statements at the top of the contract and is expected to be satisfied after executing the constructor and every public function in the contract. Although technically it can be captured using function specifications (for both the constructor and every public function), it is typically used to capture contract-level behaviors that are expected to hold always regardless of the functionalities provided

Table 1. Core features of Solidity

Func m	$m(\overline{v}) = s$
Stmt s	$s_A \mid s; s \mid$ if e then s else $s \mid$ while e do $s \mid$ require(p) $\mid$ assert(p) $\mid$ skip
Atom s_A	$v := e \mid v.m := e \mid v[e] := e$
Expr e	$l \mid v \mid v.m \mid v[e] \mid e \oplus e \mid \odot e \mid m(\overline{e})$

in the contract.

In addition, iContract supports a number of syntactic sugars which ease the writing of specification. For instance, for each function, loop, or external function call, we assume that all global variables may be modified unless a modifies($\overline{x}$) statement is put in place (e.g., function definitions, function calls), which specifies that all except those variables in $\overline{x}$ remain unchanged. Additionally, when variable x is a mapping, we allow users to write modifies($x[a]$) where a is constant value to state that only the value at location a of x is modified, while the values at other locations are not.

In terms of specifying the expected behaviors of smart contracts, our specification language has mulitple advantages over existing approaches [14,23,25]. First, our specification language is designed to avoid global reasoning with the help of call invariants. Second, the reverts_if(p) statements allow us to easily capture explicit reverts which are very common in smart contracts in the form of *require*, *revert*() and so on. Note that this feature is missed from approaches such as Solc-verify and as a result, those respective tools often generate false alarms, i.e., reporting violation of postcondition on transactions that ought to be reverted. Last, our specification is mostly based on well-known and well-founded concepts which makes it easy to adopt.

3.2 Formalization

In the following, we provide the necessary formalization of our specification language as well as smart contracts so that we can present precisely how our approach works. Note that since all our verification effort (including static verification and runtime verification) takes place at the function-level, all we need to formalize are smart contract functions and function-level specification.

Defining Smart Contracts: To ease the discussion hereafter, we model Solidity's core (function-level) features using the language presented in Table 1. A function m includes parameters $\overline{v}$, and a body statement s. A statement s is an atomic statement s_A, a conditional statement, a while loop, an assertion, revert statement, and it also can be a sequence of statements (according to the definition shown in Table 1). An atomic statement s_A is an assignment to a variable ($v := e$), an assignment to member of a variable ($v.m := e$), or an assignment to an array element ($v[e] := e$). An expression e is a literal l, a variable v, a

member access $v.m$, an index access $v[e]$, a binary expression $e \oplus e$, a unary expression $\odot\, e$, or a call $v.m(\overline{e})$ of a local function (in the same contract) or an external function (in a different contract). We use *rev* as a preserved variable for revert condition: It is true if the contract has been reverted. Note that we can simply transform other Solidity features into our core language features such as the statement $require(a)$ is equivalent to the statement i) $assert(a \wedge \neg rev)$ in verifying code against a function variant or ii) $revert(\neg a \wedge rev)$ in the verification of reverts_if(...).

To define the semantics of smart contracts, we define a set Var contains all the variables in the contract, a set Mem contains all the members of the data structures in the contract, a set of mapping for arrays A, and data structures (where $A \cap Var = \emptyset$), a set Loc contains all the memory locations, a set Val contains all non-memory values (i.e., $Val = Int \cup Float \cup Bool \cup Str$, with Int, $Float$, $Bool$, and Str are the sets containing integer, floating-point, boolean, and string literals). We use two mapping functions $S \in Stacks$ and $H \in Heaps$ to keep track of the execution environment. Consequently, a program state $\sigma_c \in States$ is defined by a pair of stack and heap, as follows.

$$
\begin{aligned}
&S \in Stacks =_{\text{def}} Var \rightarrow (Val \cup Loc)\\
&H \in Heaps =_{\text{def}} Loc \rightarrow (Type \rightarrow (Mem \cup Int) \rightarrow (Val \cup Loc))\\
&\sigma_c \in States =_{\text{def}} Stacks \times Heaps
\end{aligned}
$$

where the set $Type$ contains all the data structure types defined in the contract as well as the array type.

We define a standard small-step operational semantics of smart contracts (based on the semantics of Solidity). A configuration C is a pair (s, σ_c) where s is a program and σ_c is a program state (i.e., the valuation of both S and H). The semantics is given by a binary relation, $\rightsquigarrow$, on configurations. Its intended interpretation is that $(s, \sigma_c) \rightsquigarrow (s', \sigma_c')$ holds if the execution of the statement in the configuration (s, σ_c) can result in the new program configuration (s', σ_c'). An execution (of s) is a possibly infinite sequence of configurations $(C_i)_{i \geq 0}$ with $C_0 = (s, \_)$ such that $C_i \rightsquigarrow C_{i+1}$ for all $i \geq 0$. We define $\rightsquigarrow^*$, the reflexive-transitive closure of $\rightsquigarrow$, to capture finite executions $(C_i)_{0 \leq i \leq n}$. The details of the small step semantics is present in Fig. 2.

Defining the Specification Language: Our specification language is constituted of predicates defined using the syntax below.

$$
\begin{aligned}
&\Phi, p, q \coloneqq \Psi \mid \Phi \vee \Phi \qquad \Psi \coloneqq a \otimes a \mid \Psi \wedge \Psi\\
&\quad a \coloneqq e \mid a \oplus a \mid \odot a \;\; e \coloneqq l \mid v \mid v[a] \mid v.m \mid old(v) \mid g(v)
\end{aligned}
$$

In general, a predicate Φ is a disjunction with one or multiple conjunctions Ψ. Each conjunct in Ψ is a relational predicate with $\otimes$ is a relational operator (i.e., $>$, $\geq$, $=$, $\neq$, $<$, $\leq$). The left-hand side and right-hand side of a relational predicate are arithmetic expressions. An arithmetic expression may have one atomic expression or multiple of them connected by binary operators $\oplus$ (i.e., $+$, $-$, $*$, $/$) or unary operators $\odot$ (i.e., $\neg$, $-$). An atomic expressions includes

$$\frac{}{\langle S,H\rangle \vdash l \Downarrow l}\ \textsc{Const} \qquad \frac{S(v)=l \quad H(l)=(type(v),m,k)}{\langle S,H\rangle \vdash v.m \Downarrow k}\ \textsc{Access}$$

$$\frac{}{\langle S,H\rangle \vdash v \Downarrow S(v)}\ \textsc{Var} \qquad \frac{S(v)=l \quad S(e)=i \quad H(l)=(type(v),i,k)}{\langle S,H\rangle \vdash v[e] \Downarrow k}\ \textsc{Select}$$

$$\frac{\langle S,H\rangle \vdash e_1 \Downarrow k_1 \quad \langle S,H\rangle \vdash e_2 \Downarrow k_2}{\langle S,H\rangle \vdash e_1 \oplus e_2 \Downarrow k_1 \oplus k_2}\ \textsc{Binary} \qquad \frac{\langle S,H\rangle \vdash e \Downarrow k_1}{\langle S,H\rangle \vdash \odot e \Downarrow \odot k_1}\ \textsc{Unary}$$

$$\frac{}{\langle S,H\rangle, \text{revert}; s_2 \rightsquigarrow \langle S_0,H_0\rangle, \text{skip}}\ \textsc{Revert} \qquad \frac{}{\langle S,H\rangle, \text{skip}; s_2 \rightsquigarrow \langle S,H\rangle, s_2}\ \textsc{Skip}$$

$$\frac{\langle S,H\rangle, s_1 \rightsquigarrow \langle S_1,H_1\rangle, s_{1'}}{\langle S,H\rangle, s_1; s_2 \rightsquigarrow \langle S_1,H_1\rangle, s_{1'}; s_2}\ \textsc{Seq} \qquad \frac{\langle S,H\rangle, s_1 \rightsquigarrow \langle S_1,H_1\rangle, \text{abort}}{\langle S,H\rangle, s_1; s_2 \rightsquigarrow \langle S_1,H_1\rangle, \text{abort}}\ \textsc{Seq-Err}$$

$$\frac{\langle S,H\rangle \vdash e \Downarrow k \quad S_1 = S[v \leftarrow k]}{\langle S,H\rangle, v := e \rightsquigarrow \langle S_1,H\rangle, \text{skip}}\ \textsc{Assign-1} \qquad \frac{\langle S,H\rangle \vdash v \Downarrow k \quad k \notin dom(H)}{\langle S,H\rangle, v.m := e \rightsquigarrow \langle S,H\rangle, \text{abort}}\ \textsc{Err1}$$

$$\frac{\langle S,H\rangle \vdash v \Downarrow k \quad k \in dom(H) \quad \langle S,H\rangle \vdash e \Downarrow k_1 \quad H_1 = H[(k,type(v),m) \leftarrow k_1]}{\langle S,H\rangle, v.m := e \rightsquigarrow \langle S,H_1\rangle, \text{skip}}\ \textsc{Assign-2}$$

$$\frac{\langle S,H\rangle \vdash v \Downarrow k \quad \langle S,H\rangle \vdash e_1 \Downarrow k_1 \quad \langle S,H\rangle \vdash e_2 \Downarrow k_2 \quad H_1 = H[(k,Array,k_1) \leftarrow k_2]}{\langle S,H\rangle, v[e_1] := e_2 \rightsquigarrow \langle S,H_1\rangle, \text{skip}}\ \textsc{Assign-3}$$

$$\frac{\langle S,H\rangle \vdash v \Downarrow k \quad k \notin dom(H)}{\langle S,H\rangle, v[e_1] := e_2 \rightsquigarrow \langle S,H_1\rangle, \text{abort}}\ \textsc{Err2} \qquad \frac{\langle S,H\rangle \vdash e_1 \Downarrow k_1 \quad k_1 \notin size(v)}{\langle S,H\rangle, v[e_1] := e_2 \rightsquigarrow \langle S,H_1\rangle, \text{abort}}\ \textsc{Err3}$$

$$\frac{\langle S,H\rangle \vdash b \Downarrow True}{\langle S,H\rangle, \text{if } b \text{ then } s \text{ else } s' \rightsquigarrow \langle S,H\rangle, s}\ \textsc{If-T} \qquad \frac{\langle S,H\rangle \vdash b \Downarrow False}{\langle S,H\rangle, \text{if } b \text{ then } s \text{ else } s' \rightsquigarrow \langle S,H\rangle, s'}\ \textsc{If-F}$$

$$\frac{\langle S,H\rangle \vdash b \Downarrow True}{\langle S,H\rangle, \text{while } b \text{ do } s \rightsquigarrow \langle S,H\rangle, s; \text{while } b \text{ do } s}\ \textsc{Loop-T}$$

$$\frac{\langle S,H\rangle \vdash b \Downarrow False}{\langle S,H\rangle, \text{while } b \text{ do } s \rightsquigarrow \langle S,H\rangle, \text{skip}}\ \textsc{Loop-F}$$

$$\frac{v.m(\overline{p}) = s \quad \langle S,H\rangle \vdash \overline{e} \Downarrow \overline{k} \quad S' = S[\overline{p} \leftarrow \overline{k}]}{\langle S,H\rangle \vdash v.m(\overline{e}) \rightsquigarrow \langle S',H\rangle, s}\ \textsc{Call}$$

Fig. 2. Small-step operational semantics of the smart contract language, given by the binary relation $\rightsquigarrow$ over $Stacks \times Heaps$

a literal l, a variable v, a member access $v.m$, and an index access $v[a]$. The expression $v.m$ accesses the value stored in the member m of a struct v, whereas the expression $v[a]$ accesses the value at key a of a mapping v. In addition, we provide a function $old(v)$ which returns the value of variable v at the beginning of the function (for function specifications) or the loop (for loop invariant) or before an external function call (for call invariants). Moreover, we support a library of externally defined function $g(v)$. One example is the *sum* function, which, given a mapping v, computes the sum of all values stored in v.

$$
\begin{array}{ll}
S, H \models \Phi_1 \vee \Phi_2 & \textbf{iff } (S, H \models \Phi_1) \vee (S, H \models \Phi_2) \\
S, H \models \Psi_1 \wedge \Psi_2 & \textbf{iff } (S, H \models \Psi_1) \wedge (S, H \models \Psi_2) \\
S, H \models a_1 \otimes a_2 & \textbf{iff } (S, H \models a_1 = k_1) \wedge (S, H \models a_2 = k_2) \wedge (k_1 \otimes k_2) \\
S, H \models a_1 \oplus a_2 = k & \textbf{iff } (S, H \models a_1 = k_1) \wedge (S, H \models a_2 = k_2) \wedge (k_1 \oplus k_2 = k) \\
S, H \models \odot a = k & \textbf{iff } (S, H \models a = k_1) \wedge (\odot k_1 = k) \\
S, H \models l = l & \textbf{iff true} \\
S, H \models v = k & \textbf{iff } S(v) = k \\
S, H \models v[a] = k & \textbf{iff } S(v) \in dom(H) \wedge 0 \geq S(a) < size(v) \wedge H(S(v)) = (Array, S(a), k) \\
S, H \models v.m = k & \textbf{iff } S(v) \in dom(H) \wedge H(S(v)) = (type(v), m, k) \\
S, H \models old(v) = k & \textbf{iff } S_0, H_0 \models v = k \\
S, H \models sum(v) = k & \textbf{iff } type(v) = Array \wedge S(v) \in dom(H) \wedge \sum_{i=0}^{size(v)} \{v[i]\} = k
\end{array}
$$

Fig. 3. Specification formula semantic where $dom(f)$ returns the domain of function f, $size(v)$ the range of index of the array v.

The semantics is defined according to a satisfaction relation $S, H \models \Phi$ which is defined in a common way, as shown in Fig. 3. Next, we define the correctness in our specification language. First, regarding contract invariants, given a contract c associated with multiple cinv(p) statements, the contract is correct if and only if each ensures(p, p) is satisfied by all the public functions including the constructor. Second, regarding function specifications, given a function $m(\overline{v}) = s$ associated with multiple ensures(p, q) and reverts_if(p') statements, the function is correct iff for each ensures(p, q) statement, the following is satisfied.

$$\forall \sigma_c, \sigma_c'.\ \sigma_c \models p \wedge (s, \sigma_c) \rightsquigarrow^* (\text{skip}, \sigma_c') \implies \sigma_c' \models q$$

Furthermore, for each reverts_if(p') statement, the following is satisfied

$$\forall \sigma_c, \sigma_c'.\ \sigma_c \models p' \wedge (s, \sigma_c) \rightsquigarrow^* (\text{require(b)}, \sigma_c') \implies \sigma_c' \models \neg b$$

Third, regarding loop invariants, given a loop *while b do s* associated with an loop_inv(q) statement at the beginning, the following must be satisfied where L is a function that filters states satisfying b.

$$\forall \sigma_c, \sigma_c'.\ \sigma_c \models q \wedge (s, L(\sigma_c, b)) \rightsquigarrow^* (\text{skip}, \sigma_c') \implies \sigma_c' \models q$$

Fourth, for each achieves(p, q), the following must be satisfied.

$$\forall \sigma_c'. \exists \sigma_c.\ \sigma_c' \models q \implies \sigma_c \models p \wedge (s, \sigma_c) \rightsquigarrow^* (\text{skip}, \sigma_c')$$

Lastly, regarding call invariants, given an external function call $m(\overline{e})$ associated with multiple call_inv(p, q), for any implementation s of $m(\overline{e})$, the following must be satisfied: $\forall \sigma_c, \sigma_c'.\ \sigma_c \models p \wedge (s, \sigma_c) \rightsquigarrow^* (\text{skip}, \sigma_c') \implies \sigma_c' \models q$.

4 Verification

We present our compositional verification algorithm by first defining an encoding function $post(\sigma_i, s_i)$ and illustrating how to utilize it to validate function specification ensures(p, q) and revert specification reverts_if(p). We remark that we

abuse the notation σ_i to represent a symbolic state where its syntax is similar to our specification language. Furthermore, we provide encoding rules that substitute loops and function calls with their specifications.

4.1 Function Validation

We define an encoding function $post(\sigma_i, s_i)$ that takes a pre-state σ_i and a statement s_i as inputs, and procedure post-states σ_k as output. Given a function $m(\overline{v}) = s$ which may contain loops as well as internal and external function calls, our validations are defined as follows. A function specification ensures(p, q) with implementation s is verified if $post(p, s)$ returns σ such that $\sigma \Rightarrow q$. The execution $post(p, s)$ indicates that the encoding process starts with pre-state p. After processing the statement s, the validation formula $\sigma \Rightarrow q$ means that if the function is not reverted then the encoding starts with p implies the post-condition q. Similarly, a specification reverts_if(p) is verified if $post(p, s)$ returns the post-state $\sigma \wedge rev$ at exits. Note that the procedure of verifying ensures(p, p) utilizes the encoding rule Revert-1. On the other hand, other Revert rules, such as Revert-2 and Revert-3, are employed for the verification of reverts_if(p).

4.2 Generating Proof Obligations

We define encoding function $post(\sigma_i, s_i)$ using encoding rules, each of which is of the following form.

$$\frac{premise_0 \quad ... \quad premise_i}{\sigma_i, s_i \rightsquigarrow \sigma_k}$$

This transition rule means given a pre-state σ_i, a statement s_i, it executes $premise_0$, ..., $premise_i$ to obtain the post-state σ_k. The encoding rules are shown in Fig. 4. Note that the encoding transforms the code into the predicates supported by off-the-shelf SMT solver Z3. While most of the syntax is self-explanatory, we use the notation $v[a \rightarrow l]$ to represent an array with $v[a \rightarrow l][a'] = v[a']$ when $a' \neq a$ and $v[a \rightarrow l][a] = l$.

The rules are divided into three groups, i.e., rules for local operations, rules for external function calls, and rules for revert. Rules for local operations include Seq, Assign-1, Assign-2, Assign-3, If and Loop. They are similar to the traditional Hoare rules. In the Assign-2 and Assign-3, we substitute v before the assignment with x, and set the current v as the result of update value $e[x/v]$ to the value located at index $m[x/v]$ or property m. In the Assign-2, for each write operation to $v[m]$, we compute sum(v) by adding the current sum u to the difference between the new value e and the old value $v[m]$, i.e., $u' = u + e - v[m]$. The Loop substitutes the loop with its invariant and exiting condition (i.e., $\neg b$).

Rules for revert include Revert-1 (the non-revert condition is part of the pre-condition), Revert-2 (the revert condition met) and Revert-3 (the revert condition is not met). The idea is that the function is reverted if any of the condition leading to revert is satisfied. If the revert condition is satisfied, the

$$\frac{\sigma, s_1 \rightsquigarrow q_1 \qquad q_1, s_2 \rightsquigarrow \sigma_2}{\sigma, s_1; s_2 \rightsquigarrow \sigma_2}\ \textsc{Seq} \qquad \frac{\sigma' = \exists x.\ \sigma[x/v] \wedge v = e[x/v]}{\sigma, v := e \rightsquigarrow \sigma'}\ \textsc{Assign-1}$$

$$\frac{u' = u + e - v[m] \quad \sigma' = \exists x.\ \sigma[x/v] \wedge v = x[m[x/v] \rightarrow e[x/v]]}{\sigma \wedge \mathrm{sum(v)} = u, v[m] := e \rightsquigarrow \sigma' \wedge \mathrm{sum(v)} = u'}\ \textsc{Assign-2}$$

$$\frac{}{\sigma \wedge rev, s \rightsquigarrow \sigma \wedge rev}\ \textsc{Rev-Prop} \qquad \frac{\sigma' = \exists x.\ \sigma[x/v] \wedge v = x[m \rightarrow e[x/v]]}{\sigma, v.m := e \rightsquigarrow \sigma'}\ \textsc{Assign-3}$$

$$\frac{\sigma \wedge b, s_0 \rightsquigarrow \sigma_1 \qquad \sigma \wedge \neg b, s_1 \rightsquigarrow \sigma_2}{\sigma, \text{if } b \text{ then } s_0 \text{ else } s_1 \rightsquigarrow \sigma_1 \vee \sigma_2}\ \textsc{If} \qquad \frac{\sigma \Rightarrow q}{\sigma, \mathrm{assert(q)} \rightsquigarrow \sigma}\ \textsc{Assert}$$

$$\frac{\sigma' = \sigma \wedge p}{\sigma, require(p) \rightsquigarrow \sigma'}\ \textsc{Revert-1} \qquad \frac{\sigma \Rightarrow \neg p}{\sigma, require(p) \rightsquigarrow \sigma \wedge rev}\ \textsc{Revert-2}$$

$$\frac{\sigma \not\Rightarrow \neg p}{\sigma, require(p) \rightsquigarrow \sigma}\ \textsc{Revert-3}$$

$$\frac{\sigma' = \exists \overline{x}.\ \sigma[\overline{x}/\overline{v}] \wedge p \wedge \neg b}{\sigma, \mathrm{modifies}(\overline{v}); \mathrm{loop\_inv(p)}; \text{while } b \text{ do } s \rightsquigarrow \sigma'}\ \textsc{loop}$$

$$\frac{\mathrm{reverts\_if(p)} \in \textsc{spec}(m(\overline{e})) \quad \sigma, require(\neg p); m(\overline{e}) \rightsquigarrow \sigma'}{\sigma, m(\overline{e}) \rightsquigarrow \sigma'}\ \textsc{Revert-Inter}$$

$$\frac{\sigma \Rightarrow p \qquad \sigma' = \exists \overline{x}.\ \sigma[\overline{x}/\overline{v}] \wedge q}{\sigma, \mathrm{modifies}(\overline{v}); finv(p, q); m(\overline{e}) \rightsquigarrow \sigma'}\ \textsc{Call-Spec}$$

Fig. 4. Encoding rules (where $finv(p, q)$ is ensures(p, q) or call_inv(p, q)) (Color figure online)

value of *rev* is set, and after that our system skips all the remaining statements by using rule Rev-Prop.

Rules for external function calls include Call-spec, which replaces function calls using either function specifications (if it calls for a local function) or call invariants (if it calls for an external function). This rule updates modified variables $\overline{v}$ through the substitutions $\sigma[\overline{x}/\overline{v}]$. Note that, to propagate the reverts_if(p) back to the caller, via rule Revert-Inter, we simply convert it to $require(\neg p)$ before the function call is encoded. Moreover, each ensures(p, q) is lifted to the context of the current function by substituting free variables appearing on parameters with their corresponding arguments.

Note that the correctness specification may be over-approximating and thus our verification may lead to false alarms and spurious counterexamples. Instead of running test cases with extra costs, the incorrectness specification associated with external function calls is used to construct counterexamples. According to the concrete values from counterexamples, we first determine an execution path leading to the violation, and then use the achieves(p, q) statements associated with the involved external function calls to check whether the counterexample is real. We develop another predicate $post_U(p, s)$ to compute under-approximating post-states for the implementation s, then our system confirms the bug described in the spec if $q \Rightarrow \sigma$. In term of encoding for $post_U$, dropping execution paths is

allowed in incorrectness logic. Therefore, the number of loop iterations can be freely chosen. Only the true-branch or the false-branch is selected while encoding an if-statement. If there is an execution path that satisfies the incorrectness specification then the counterexample is determined to be a real violation. Finally, to handle function call $m(\overline{e})$; modifies($\overline{v}$); achieves(p', q') at the calling states σ, it first tests $p' \Rightarrow \sigma$. If this test succeeds, it produces $\sigma[\overline{x}/\overline{v}] \wedge q'$ as the post-states. Otherwise, if $\overline{v} \cap \textsc{FreeVars}(\sigma) = \emptyset$, it checks $\textsc{Sat}(p' \wedge \sigma)$. If it is satisfied, it produces $\sigma[\overline{x}/\overline{v}] \wedge q'$ as the post-states. The soundness of the former comes from Consequence rule and the later is from Constancy rule in incorrectness logic.

5 Implementation and Evaluation

5.1 Implementation

iContract is implemented with around 1K lines of Python code. It supports most features of Solidity version 0.5.1 including inheritance and important built-in functions (e.g., *send*, and *call*). iContract uses a locally installed Solidity compiler to compile a user-provided Solidity file into a JSON file containing the typed abstract syntax tree (AST). Then, iContract analyzes the AST to encode contracts into predicates using the Z3 library. We leverage NatSpec [4] format to define our own specifications.

The encoding is mostly straightforward except some relevant details that we discuss below. We use SMT Integer to model int/unsigned and int/address and so on[2]. To support contract inheritance, we implement a symbol table which allows us to query global variables and functions of parent contracts using inheritance tree provided by the Solidity compiler. We also take into account function overriding and variable hiding.

Our current implementation has several limitations. First, it does not support low-level API calls including inline assembly, Application Binary Interface functions, and bitwise operations. Second, iContract does not compute gas consumption to determine out-of-gas exceptions. Last, iContract analyzes contracts without the presence of aliasing. Note that although Solidity allows two variables reference to the same data location (i.e., aliasing), it is not very common in Solidity and we leave it to future work.

5.2 Experimental Evaluation

In the following, we design and conduct multiple experiments to answer the following research questions (RQ).

- RQ1: Can iContract verify real-world smart contracts?
- RQ2: How does iContract compare with Solc-verify [14], a state-of-the-art tool for verifying function-level properties?

[2] Note that runtime checking for arithmetic overflow has been introduced since Solidity 0.8 and thus no longer an issue.

Table 2. Statistics on verified contracts

Project	#Contracts	#Functions	#Ifs	#Specifications	LOC	#Transactions (mil)
BAT	4	16	20	17	179	3.97
BNB	2	13	22	25	150	1.00
HT	4	13	4	2	127	0.67
HOT	3	22	29	28	279	0.95
IOTX	8	32	28	35	500	0.28
QNT	5	24	13	16	239	1.21
MANA	11	28	21	70	282	2.50
ZIL	9	35	42	70	353	0.44
NXM	3	37	36	40	448	0.12
SHIB	4	33	12	33	448	9.5

RQ1 aims to evaluate whether iContract is useful for some practical smart contracts. RQ2 aims to evaluate whether iContract's approach (in particular, its specification language) can achieve its goals better than existing approaches.

In the following, we present the evaluation results and answer the questions. All our experiments are conducted on a single processor running an Ubuntu 16.04.6 LTS machine with Intel(R) Core(TM) i9-9900 CPU @ 3.10 GHz and 64 GB of memory. The timeout is set to be 5 min for verifying the specification. Our implementation and the verified contracts are available online [2].

RQ1: To answer this question, we identify a set of 10 high-profile projects from EtherScan [3]. The relevant statistic of these contracts is shown in Table 2. The table shows the name of each project. For each project, it shows the number of contracts (#Contracts), the number of functions (#Functions), the number of if-statements (#Ifs), the number of specification statements (#Specifications), line of codes (LOC), and the number of transactions (#Transactions) in millions. Most of them have over 200 lines of code and 20 functions. Each project is associated with a Solidity file, which typically contains multiple contracts including a main one as well as library or parent contracts. Since not all smart contracts are written in the same Solidity versions, we have to convert them to a fixed version (i.e., 0.5.1). This is necessary to ensure the consistency of our verification results. All specifications are manually written by the authors and directly injected into the Solidity files. The specifications are written in such a way that they describe the logic of each function as precise as possible. There are 124 reverts_if(), 2 contract invariants, 4 call invariants, 206 function specifications.

The verification results for each project is shown in Table 3 under column iContract. The column #V shows the number of specifications that were successfully verified. The column #F shows the number of falsified specifications. The column Time shows the average verification time in seconds. Since we group our specifications into a single specification to compare with Solc-verify, the column #Sp is less than the one shown in Table 2. Most of the projects are

```
1 /// reverts_if(_amount == 0);
2 function mint(address _to, uint256 _amount) private {
3   // Guard against overflow
4   require(balances[_to] + _amount > balances[_to]);
5   balances[_to] = balances[_to].add(_amount);
6 }
```

Fig. 5. An example illustrating the effectiveness of reverts_if() in identifying incorrect require statements

verified within 5 s. Among 336 specification statements, 3 of them are falsified. After manually investigating them, we confirm that iContract exposes contract invariant violations in HOT, QNT and BNB. First, BNB stores the frozen tokens in a mapping called *freezeOf*. When tokens are frozen, they are not subtracted from *totalSupply*. As a result, sum(*balances*) $\neq$ *totalSupply*. Second, the *totalSupply* of QNT remains unchanged even when refresh QNT is created by calling the function *mint*. Again, sum(balances) $\neq$ *totalSupply*. Third, as shown in Fig. 5, HOT has the following *require* statement at line 4 which is meant to prevent overflow according to the documentation. However, it also prevents non-overflow cases such as when $\_amount = 0$.

RQ2: To answer RQ2, we compare iContract against Solc-verify, a state-of-the-art tool for verifying function-level properties of smart contracts [14]. Solc-verify is selected as it shares much similarity with iContract, i.e., it supports contract, function and loop invariants. Other verifiers either do not support user-defined specification (such as Verismart [24]), or restrict their specification in specific forms (e.g., linear temporal logic such as in Verx [23] and SmartPulse [25]), which are not expressive enough to capture the specification required to verify the correctness of the contracts used in our experiments. We first translate all specifications written in our language to the ones supported by Solc-verify. The translation is not straightforward due to the fact that Solc-verify does not support reverts_if(p) and call invariants. We thus remove the call invariants, reverts_if(p) and convert our ensures(p, q) statements into Solc-verify's specifications. The results are summarized in Table 3 under the column *Solc-verify*. While Solc-verify does verify most of the contracts, results inconsistent with ours are reported for 3 contracts, as shown in column #Consistent. All of them are falsified by Solc but are verified by iContract. Our investigation shows that the reason is the missing specifications for external function calls, i.e., the call invariants. In the ZIL project, the external function call *token.transfer* (*owner*, *amount*) transfers tokens to the *owner*. Solc-verify assumes that all global variables are modified after the call and thus sum(balances) = *totalSupply* is falsified. In contrast, our call invariants indicate that the variable *balances* is unchanged and the specification sum(balances) = *totalSupply* is preserved. In the BAT and BNB projects, well-known external functions call such as *send*() and *transfer*() are not properly handled in Solc-verify. We remark that besides supporting specification features such as reverts_if(p) and call invariants, iContract works on Solidity code directly without converting it to another language

Table 3. Comparison against Solc-verify

Project	# Sp	iContract			Solc-Verify			
		#V	#F	Time (s)	#V	#F	Consistent	Time (s)
BAT	13	13	0	4.93	12	1	×	4.51
BNB	15	14	1	7.20	13	2	×	4.00
HT	2	2	0	1.10	2	0	✓	2.77
HOT	17	17	0	1.41	17	0	✓	4.38
IOTX	23	23	0	2.09	23	0	✓	4.26
QNT	11	10	1	1.33	10	1	✓	4.69
MANA	42	42	0	3.94	42	0	✓	6.63
ZIL	40	40	0	4.61	39	1	×	7.13
NXM	22	22	0	2.24	22	0	✓	6.31
SHIB	23	23	0	1.76	23	0	✓	5.95

for verification. This makes verification of the falsified specification statements straightforward in iContract, i.e., by transforming the respective undefined functions into assertions.

6 Related Work and Conclusion

The verification for smart contracts has been the interests of multiple researchers. The systems that are closely related to ours are Solc-verify [14] and MVP [9]. Solc-verify translates Solidity contracts into the Boogie intermediate language, and relies on the Boogie system for verification. It supports contract invariant, loop invariant, and pre-/post conditions. In particular, Solc-verify assertion language targets the safety of low-level properties (e.g., the absence of overflows) and high-level contract invariants (e.g., the sum of user balances equates to the total supply). Similarly, Dill *et al.* recently proposed MVP, a static verifier based on the Boogie verifier, for smart contracts in the Move language [9]. MVP supports both contract invariants and functional invariants via pre/post conditions. It also generates global invariants for runtime checking. MVP enables an alias-free memory model through reference elimination which relies on borrow semantics. MVP was deployed for continuous verification on Move code and Diem blockchain. iContract supports all the features supported by Solc-verify and MVP, and additionally supports features like revert and call invariants that are designed to handle dynamic dispatching on unknown function calls.

There are several other verification systems for smart contracts developed in the last few years, e.g., VeriSmart [24], SmartACE [29], and VerX [23]. VeriSmart [24] focuses on intra-procedural analysis for verifying arithmetic (over- and under-flows) safety. The main contribution of their work is an algorithm that could refine transaction invariants of arbitrary transactions. These invariants boost the precision of such verification. However, VeriSmart lacks inter-

procedural reasoning. SmartACE [29] is a framework that can verify user-annotated assertions by running multiple independent analysers. It models smart contract library and transforms the verification problem into off-the-self analysers like constrained Horn clause solving (e.g., SeaHorn) for correctness verification. In contrast, iContract presents a built-in static analyser for a rich specification. Finally, VerX [23] focuses on temporal properties of Ethereum contracts. It reduces the temporal safety verification to reachability verification and applies the state-of-the-art reachability checking technique. While temporal logic based specification is useful for specifying global properties, we believe that our specification language is better for supporting the motto of "specification is law" and has its advantage on compositional verification.

To conclude, in this work, we design a static verification system with a specification language which supports fully compositional verification. Using iContract, we have verified 10 high-profile smart contracts against manually developed detailed specifications, many of which are beyond the capacity of existing verifiers. In the future, we intend to improve the performance of iContract further with optimization techniques.

References

1. Bamboo: a language for morphing smart contracts. https://github.com/pirapira/bamboo
2. Dataset. https://anonymous.4open.science/r/zero1-0DEE/
3. Etherscan. https://etherscan.io/
4. Natspec format. https://docs.soliditylang.org/en/v0.8.17/natspec-format.html
5. Solidity - Solidity documentation. https://docs.soliditylang.org/en/stable/
6. swcregistry. https://swcregistry.io/
7. Vyper - Vyper documentation. https://docs.vyperlang.org/en/stable/
8. Daian, P.: DAO exploit. https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/
9. Dill, D., Grieskamp, W., Park, J., Qadeer, S., Xu, M., Zhong, E.: Fast and reliable formal verification of smart contracts with the move prover. In: TACAS 2022. LNCS, vol. 13243, pp. 183–200. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_10
10. Ernst, M.D., et al.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)
11. Fabian Vogelsteller, V.B.: EIP-20: token standard, November 2015. https://eips.ethereum.org/EIPS/eip-20
12. Ferreira Torres, C., Jonker, H., State, R.: Elysium: context-aware bytecode-level patching to automatically heal vulnerable smart contracts. In: Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses, pp. 115–128 (2022)
13. Grieco, G., Song, W., Cygan, A., Feist, J., Groce, A.: Echidna: effective, usable, and fast fuzzing for smart contracts. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 557–560 (2020)
14. Hajdu, Á., Jovanović, D.: solc-verify: a modular verifier for solidity smart contracts. In: Chakraborty, S., Navas, J.A. (eds.) VSTTE 2019. LNCS, vol. 12031, pp. 161–179. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-41600-3_11

15. Li, A., Choi, J.A., Long, F.: Securing smart contract with runtime validation. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 438–453 (2020)
16. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 254–269 (2016)
17. Mariano, B., Chen, Y., Feng, Y., Lahiri, S.K., Dillig, I.: Demystifying loops in smart contracts. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 262–274 (2020)
18. Mueller, B.: Smashing ethereum smart contracts for fun and real profit. HITB SECCONF Amsterdam **9**, 54 (2018)
19. Nguyen, T.D., Pham, L.H., Sun, J.: SGUARD: towards fixing vulnerable smart contracts automatically. In: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24–27 May 2021, pp. 1215–1229. IEEE (2021). https://doi.org/10.1109/SP40001.2021.00057
20. Nguyen, T.D., Pham, L.H., Sun, J., Lin, Y., Minh, Q.T.: sFuzz: an efficient adaptive fuzzer for solidity smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 778–788 (2020)
21. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL) (2019). https://doi.org/10.1145/3371078
22. Palladino, S.: The parity wallet hack explained, July 2017. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/
23. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: VerX: safety verification of smart contracts. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 1661–1677 (2020). https://doi.org/10.1109/SP40000.2020.00024
24. So, S., Lee, M., Park, J., Lee, H., Oh, H.: VERISMART: a highly precise safety verifier for ethereum smart contracts. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 1678–1694 (2020)
25. Stephens, J., Ferles, K., Mariano, B., Lahiri, S., Dillig, I.: SMARTPULSE: automated checking of temporal properties in smart contracts. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 555–571. IEEE (2021)
26. Szabo, N.: Formalizing and securing relationships on public networks. First Monday (1997)
27. Torres, C.F., Iannillo, A.K., Gervais, A., State, R.: ConFuzzius: a data dependency-aware hybrid fuzzer for smart contracts. In: 2021 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 103–119. IEEE (2021)
28. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M.: Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 67–82 (2018)
29. Wesley, S., Christakis, M., Navas, J.A., Trefler, R., Wüstholz, V., Gurfinkel, A.: Verifying Solidity smart contracts via communication abstraction in SmartACE. In: Finkbeiner, B., Wies, T. (eds.) VMCAI 2022. LNCS, vol. 13182, pp. 425–449. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-94583-1_21
30. Wood, G., et al.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Proj. Yellow Paper **151**(2014), 1–32 (2014)