A Framework of Runtime Monitoring for Correct Execution of Smart Contracts

R. K. Shyamasundar(✉)

Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, Mumbai 400076, India
rkss@cse.iitb.ac.in

Abstract. Smart contracts have been subjected to several attacks that have exploited various vulnerabilities of languages like Solidity, which has resulted in huge financial losses. The functioning and deployment of smart contracts are somewhat different from classical programming environments. Once a smart contract is up and running, changing it, is very complicated and nearly infeasible as the contract is expected to be immutable when created. If we find a defect in a deployed smart contract, a new version of the contract has to be created and deployed with concurrence from the stakeholders. Further, when a new version of an existing contract is deployed, data stored in the previous contract does not get transferred automatically to the newly refined contract. We have to manually initialize the new contract with the past data which makes it very cumbersome and not very trustworthy. As neither updating a contract nor rolling back an update is possible, it greatly increases the complexity of implementation and places a huge responsibility while being deployed initially on the blockchain.

The main rationale for smart contracts has been to enforce contracts *safely* among the stakeholders. In this paper, we shall discuss a framework for runtime monitoring to prevent the exploitation of a major class of vulnerabilities using the programmers' annotations given in the smart contracts coded in Solidity. We have chosen several phrases for annotation mimicking declarations of concurrent programming languages so that the underlying run-time monitors can be automatically generated. The annotations simply reflect the intended constraints on the execution of programs relative to the object state relative to observables like method calls, exceptions, etc. Such a framework further adds to the advantage of debugging at the source level as the original structure is preserved and also enhances the trust of the user as the run-time monitoring assertion logs provide a rough *proof-outline* of the contract.

Keywords: Smart contract · Blockchain · EVM · Correctness · Debugging · Distributed programming languages · Object state

1 Introduction

The blockchain platform is a complex system consisting of nodes, network devices, authentication services, dApps, wallets, web interfaces, key storage,

S. Chen et al. (Eds.): ICBC 2022, LNCS 13733, pp. 92–116, 2022.
https://doi.org/10.1007/978-3-031-23495-8_7

miners, etc. It is a network of centralized and decentralized system components. Securing such a configuration is quite complex. The core technology of blockchain platforms is strongly secure as it satisfies properties like immutable, distributed, and public. However, in various use-case scenarios of the platforms, in particular dApps, the system is always a network of systems, and hence, it suffers from classic threats like data spoofing, data tampering, denial of service, privilege escalation, data-disclosure, neutralization of non-repudiation, etc. Security risks of such a system need to be treated in the traditional way. However, blockchain platforms evolving from Bitcoin have become highly flexible and expressively powerful due to the richness of underlying smart contracts. Smart contract languages today are derived from extensions of general-purpose languages like Javascript. While such a similarity makes smart contract languages look familiar to software developers, it is inadequate to accommodate domain-specific requirements of digital contracts. Smart contracts have not only shed light on the benefits of digital contracts but also on their potential risks. Smart contract languages like Solidity [30], GO [4], Clarity[1], Rust[2], DAML[3] etc., are widely used in practice. Like all software, smart contracts can contain bugs and their vulnerabilities can be exploited which can have direct financial consequences. Thus, it is very important to have a sound methodology, that is practical enough for use by a large community of smart contract programmers to check contracts for crucial properties. Quite a number of security vulnerabilities in smart contracts over Ethereum have been discovered [5] over the years and have been exploited extensively causing huge losses.

There has been a significant amount of work done in analyzing the correctness of smart contracts. A brief survey of the various approaches is given in Sect. 2.1. Broadly, the approaches use the following techniques to analyze the correctness of smart contracts: (1) symbolic execution-based static analysis/verification, (2) formal verification of smart contracts, (3) restricting the power of smart contract languages, (4) algorithmic analysis of the source- and the object-code (EVM), and (5) guidelines for the best practices. From a general programmer's perspective, it may be observed that (a) formal specification is hard; in particular, for dynamically-bound languages that require run-time checks on object's states, (b) the user prefers to debug at the source level rather than at the object level, (c) algorithmic analysis is limited, approximate, and has severe limitations due to over-/under-approximation, (d) several machine learning analyses have been post-detection and have a good amount of false positives, and (e) while the best practices serve the community extremely well, the lack of automatic tools to ascertain certain subtle advice comes in the way of gaining trust.

In this paper, we propose a framework for run-time monitoring through programmers' simple annotations that provide a trusted approach to prevent a major class of vulnerabilities from being exploited. Further, the approach has proposed a set of standard annotations that are similar to the declarative phrases

[1] https://book.clarity-lang.org.
[2] https://docs.casperlabs.io/dapp-dev-guide/writing-contracts/rust/.
[3] https://docs.daml.com/daml/reference/interfaces.html.

used in concurrent programming languages, which enables the prevention of a large number of vulnerabilities from being exploited. The main contributions of the paper are summarized below:

- Use of run-time monitoring to assure safe execution of smart contracts in Solidity, so that unintended executions with respect to the semantics of Solidity are prevented along with succinct error messages.
- Choice of annotation phrases parallels those that are used in the declarative part of a concurrent programming language and allows the prevention of a majority of vulnerabilities that are widely exploited. Such abstraction allows adapting source language compilations techniques to run-time monitoring.
- Use of run-time monitoring annotations to specify additional expressive constraints on complex/very expressive contracts.
- Automatic generation of run-time monitors from the given standard annotations and integrating the same with the smart contract source.
- Transforming the smart contract in Solidity with annotations to Solidity without annotations, preserving the semantics - leading to the capability of debugging the contract at the source level itself.

The rest of the paper is organized as follows: Sect. 2, provides an overview of smart contract vulnerabilities and a brief survey of the existing approaches to mitigate various vulnerabilities. Section 3 provides the salient aspects of the run-time monitoring framework. In Sects. 4 and 5, we discuss attack scenarios exploiting vulnerabilities followed by illustrations as to how such vulnerabilities are detected through run-time monitoring. In Sect. 6, we discuss how a majority of the vulnerabilities can be detected through standard annotations; in Sect. 6.3, it is shown how run-time monitors can be generated from the specified annotations for the given smart contract in Solidity. A comparative evaluation of the framework with those in the literature is carried out in Sect. 7. This is followed by conclusions in Sect. 8.

2 An Overview of Vulnerabilities in Smart Contracts

Serious fraud and the experience of smart contracts over the years have lead to the detection of several vulnerabilities. A spectrum of vulnerabilities that arise due to artifacts of Solidity, EVM, Blockchain structure, implicit distributed execution on a blockchain, etc., are discussed by several authors [5,32]. Vulnerabilities have been broadly classified into the following categories [5]:

1. Solidity Source Language Vulnerabilities: Some of the major vulnerabilities in this category are: re-entrancy, call to the unknown, gasless send, exception disorders, type casts, and transaction order dependence (partly due to the underlying blockchain), etc.,

2. Object Code (EVM) Vulnerabilities: Typical vulnerabilities[4] in this class are *ether* lost in the transfer, immutable bugs, etc., and
3. Blockchain-based Vulnerabilities: Major vulnerabilities in this class are generating randomness, unpredictable state, time constraints, etc.

2.1 Overcoming Smart Contract Exploits: A Brief Survey

There has been a significant amount of work in analyzing smart contracts for robustness. We can broadly classify the approaches into:

1. Static analysis based approaches [25,27,36]: Note that the characterization of vulnerabilities is quite often based on anecdotal incidents reported publicly. Luu et al. [25] present a symbolic execution tool for analyzing smart contracts at the bytecode level. The system requires an analysis of best-effort manual analysis of the EVM bytecode to characterize the security bugs in the program as a trace using constraint solver $Z3$ to eliminate infeasible traces. The tool leads to various false positives and a recent analysis [16] has shown that the verification conditions generated by it are neither sound nor complete. MPro [36] is another efficient and scalable tool that combines symbolic execution and data dependency analysis thus reducing false positives.
2. Transformation of Smart contracts: One of the early works [6], uses a two-way language-based approach for verifying smart contracts. They translate the contracts written in a subset of EVM to F*-a functional programming language with the aim of program verification. The subset is quite limited and there is no evidence that it can capture a large fraction of real-life smart contracts. Securify [35] is a lightweight security verifier for Ethereum smart contracts. It uses a new domain-specific language, Securify, that enables users to express new vulnerability patterns as they emerge. To check these patterns, Securify symbolically encodes the dependence graph of the contract in stratified Datalog and leverages off-the-shelf Datalog solvers for efficient analysis of the code. Securify derives semantic facts inferred by analyzing the contracts dependency graph and uses these facts to check a set of compliance and violation patterns. Based on the outcome of these checks, Securify classifies all contract behaviors into violations, warnings, and compliant ones. Again, there is a lot of expertise required in formulating patterns and as such arriving at patterns of complex interaction is not quite simple for a naive programmer.
3. Formal correctness, formal semantic frameworks for Solidity cum EVM: Approaches on these lines are explored in [2,10,13,16,20,21,29]. In [10] an approach for proof-carrying smart contracts is explored through an example of ERC20 treating a smart contract as a distributed program. In a theorem-proving-based approach, one is concerned with the verification of the semantic specification of EVM bytecode of the contract using several standard frameworks like *Coq*, *Isabelle*, *Why3* etc. In [15], the authors propose techniques

[4] Compiler limits like *Stack size limit* can also be exploited by adversaries but the current compilers have overcome such exploits.

and tools for a formal characterization of smart contracts and static analysis of EVM bytecode. Another interesting tool is the automated static analyzer called eThor [31] for EVM bytecode that is shown to be sound. VERX [29] is another automated verifier of functional requirements for Ethereum smart contracts.

4. New sound programming languages: [8] introduces Featherweight Solidity, a calculus formalizing the core features of the Solidity language, thus providing a fundamental step to reason about the safety properties of smart contracts' source code. Such an approach prevents some errors whereas many others, such as accesses to a nonexisting function or state variable, are only detected at the run-time and cause interruption and roll-back of transactions. A linear logic-based approach has been envisaged in [9] for abstracting resource-aware session types in the specification of smart contracts.
5. Platforms for good (defensive!) practices: Manticore[5] is an open-source tool that can explore the reachability of states having its own method of specification. Further, there has been a spectrum of guidelines/practices from industries in building smart contracts. One such approach is DappGuard [24] which provides an initial design for live monitoring based on the authors' findings on various smart contracts. MYTHRIL [26] is a service platform for Ethereum to assist programmers to avoid costly errors.

A qualitative comparative evaluation is carried out in Sect. 7.

3 Runtime Monitoring Framework: A Rationale

In formal verification of imperative programs, one establishes correctness statically by showing that the program execution satisfies the specification in all the plausible execution paths relative to the specification that include pre-/post-conditions, and invariants/assertions at various control points of the program. In the context of declarative languages, while partial correctness is implicit in the program itself, it often requires proof of termination [23]. Solidity is a rich expressive imperative language used for writing smart contracts that naturally depends on dynamic binding due to the underlying blockchain structure. While formal specification/verification of distributed programs is itself quite complex, the dynamic binding of variables and the functions/methods that get executed on the blockchain makes it much more complex. Our run-time framework is structured to monitor smart contracts using the following rationale:

1. Quite a number of vulnerabilities arise due to the dynamic nature of method binding including fallback functions. Our approach is to use run-time monitors to eliminate incorrect execution paths in line with the semantics of Solidity.
2. Run-time monitoring to control the execution traces in alignment with the sequential requirement of execution of the underlying distributed program.

[5] GitHub - trailofbits/manticore: Symbolic execution tool.

3. Run-time monitor for exceptions by appropriately composing SafeMath for general expressions.
4. Run-time monitor for user-defined assertions to satisfy the requirements of smart contracts.

In the next section, we shall illustrate the above rationale through an illustration of the detection of a few major vulnerabilities and their mitigation through run-time monitoring.

4 Run-Time Monitors to Prevent Unintended Execution Traces

In this section, we shall look at two major vulnerabilities (a) Re-entrancy and (b) Transaction order indeterminacy that has been responsible for siphoning a sizable amount of crypto-currency.

4.1 Attacks Due to Re-entrancy Vulnerability

The property of non-re-entrancy can be interpreted as follows: *When a non-recursive function is invoked, it cannot re-enter before its termination. This is a requirement, a programmer takes it for granted due to the sequentiality and atomicity requirements of transactions. But this is not always true due to the fallback mechanism.* The well-known DAO attack which happened to steal a large amount of money was due to the re-entrancy vulnerability. The attack comes in two variants that are illustrated below.

SimpleDAO: Attack1. Consider the simple DAO shown in Fig. 1 and the attacker code shown in Fig. 2. Steps are given below that capture the scenario of the attack:

1. Publish contract Mallory2 with address “xyz”.
2. Assume Mallory2 donates 100 ethers to SimpleDAO i.e., credit[“xyz”] = 100.
3. Let the adversary account be at address “123”.
4. “123” transfers 100 ethers to “xyz”. This invokes the fallback function of Mallory, which internally invokes withdraw function of DAO with amount = 100.
5. msg.sender.call.value(amount)() invokes the fallback function of Mallory2.
6. The credit of Mallory2 will never be changed and thus, all the money from DAO will be transferred to Mallory2 (even if Mallory2 invested only 100 Ethers in DAO).
7. This recursive execution will continue till: a) the gas is exhausted, or b) the balance of DAO becomes 0.

```
contract SimpleDAO {
    mapping (address => uint256) public credit;

    constructor() payable public {
        donate();
    }

    function donate() payable public{
        credit[msg.sender] += msg.value;
    }

    function withdraw(uint256 amount) public{
        if (credit[msg.sender]>= amount) {
            msg.sender.call.value(amount)("");
            credit[msg.sender]-=amount;
        }
    }

    function queryCredit(address to) public view returns (uint256) {
        return credit[to];
    }
}
```

Fig. 1. Original SimpleDAO

```
contract Mallory2 {
  SimpleDAO public dao;
  address payable owner;

  constructor(SimpleDAO addr) public payable{
    owner = msg.sender;
    dao = addr;
  }

  function attack() public payable{
    dao.donate.value(1)();
    dao.withdraw(1);
  }

  function getJackpot() public{
    dao.withdraw(address(dao).balance);
    owner.transfer(address(this).balance);
  }

  function() external payable{
    dao.withdraw(1);
  }
}
```

Fig. 2. Attacker contract Mallory2

The attack was possible due to the fact that `withdraw()` - a non-recursive function that was expected to be non-re-entrant, re-entered itself before termination due to the fallback call[6]. The execution violated the underlying semantics and succeeded in getting into the attacker's fallback function. In fact, Solidity document [12] gives the following warning: "Any interaction with another contract imposes a potential danger, especially if the source code of the contract is not known in advance. The current contract hands over control to the called contract and that may potentially do just about anything. Even if the called contract inherits from a known parent contract, the inheriting contract is only required to have a correct interface. The implementation of the contract, however, can be completely arbitrary and thus, poses a danger. In addition, be prepared in case it calls into other contracts of your system or even back into the calling contract before the first call returns. This means that the called contract can change the state variables of the calling contract via its functions. Write your functions in a way that, for example, calls to external functions happen after many changes to state variables in your contract so that your contract is not vulnerable to an exploit". It must be noted that simpleDAO is already on the blockchain, and hence, it is not feasible to predict future user contracts like Mallory2.

```
1 contract Mallory3 {
2 SimpleDAO public dao = SimpleDAO(0x818EA...);
3 address owner; bool performAttack = true;
4
5 function Mallory3(){ owner = msg.sender; }
6
7 function attack() {
8 dao.donate.value(1)(this);
9 dao.withdraw(1);
10 }
```

```
1 function() {
2 if (performAttack) {
3 performAttack = false;
4 dao.withdraw(1);
5 }}
6
7 function getJackpot(){
8 dao.withdraw(dao.balance);
9 owner.send(this.balance);
10 }}
```

Fig. 3. Mallory3 contract for Attack2

SimpleDAO: Attack2. Now consider the SimpleDAO shown in Fig. 1 and the attacker *Mallory3* shown in Fig. 3. The second attack described in [5] is more subtle and allows an adversary to steal all the *ether* from SimpleDAO, with only two calls to the fallback function. Such an attack is realized through a new contract: Mallory3 contract shown in Fig. 3 where one donates just 1 *wie* to the SimpleDAO contract. After donating 1 *wie*, the attacker can then `withdraw` 1 *wie* using `withdraw` function. The `withdraw` function sends 1 *wie* back and calls *fallback* function of Mallory3 contract, which calls *withdraw* function again. After the second call, the stack begins to unwind. While unwinding first the balance

[6] The requirement for run-time checks for overcoming the re-entrancy vulnerability follows from the decidability issues of the problem of effective call-back free contracts discussed in [17] where the general problem is shown to be undecidable in Turing-complete languages.

is updated by 0 (zero) and the second time by $2^{256} - 1$ *wei* due to underflow. This allows attackers to steal everything from SimpleDAO. Finally, Mallory3 invokes getJackpot, which steals all the ether from SimpleDAO, and transfers it to Mallory3.

4.2 Mitigation for Attack1 on SimpleDAO

```
string[] callStack; //Injected code

function withdraw(uint256 amount) public{
    checkReentrancy("withdraw"); //Injected code
    callStack.push("withdraw"); //Injected code

    if (credit[msg.sender]>= amount) {
        msg.sender.call.value(amount)("");
        credit[msg.sender]-=amount;
    }

    delete callStack[callStack.length-1]; //Injected code
    callStack.length--; //Injected code
}

//Injected function
function checkReentrancy(string memory functionName) public {
    uint flag;
    if(callStack.length > 0){
        for(uint i=callStack.length; i>0; i--) {
            if(keccak256(abi.encodePacked(callStack[i-1]))==
            keccak256(abi.encodePacked(functionName))) {
                flag = 0;
                break;
            }
        }
    } else {
        flag = 1;
    }

    require(flag==1, "Reentrant!!");
}
```

Fig. 4. Modified SimpleDAO to overcome re-entrancy (Attack1)

In the simpleDAO problem, the function "withdraw" is expected to be non-re-entrant. One simple approach is to instrument the program with run-time checks so that the re-entrancy of "withdraw" is detected and prevented from executing further. The transformed program with only the relevant code snippet is shown

in Fig. 4 which introduces run-time checks in the original DAO program shown in Fig. 1. The modified instrumentation is given below:

1. The new injected function `checkReentrancy` checks for re-entrancy of a call by book-keeping the call-stack for re-entrant calls; the re-entrancy is detected by flag through the `require` statement in `checkReentrancy`.
2. In functions `withdraw`, the `callstack.push` is introduced to keep track of the new call and the call stack is popped out on exiting the procedure, and a flag is set on re-entrancy.
3. It is to be noted the monitors are checking the conditions on the object state at run-time.

4.3 Mitigation for Attack2 on SimpleDAO

From the description given in Sect. 4.3, and Fig. 3, we need to overcome the following two issues:

1. Re-entrancy of `withdraw`: The remedial measure for the re-entrancy can be taken on the lines of the mitigation shown in Fig. 4.
2. Exception Handling for overflow/underflow while reducing the amount after getting the funds back can be done using the technique of SafeMath [28] for recent versions of Solidity[7]. For instance, in the context of unsigned integer arithmetic, the code snippet for subtraction is given below:

```
function sub(uint256 a, uint256 b) internal pure returns (unint256) {
  require(b<= a, "safemath:Subtraction Overflow");
  uint256 c = a -b;
  return c;
  }
```

3. In context of context of `withdraw`, its last line gets replaced by

```
credit[msg.sender ]= SafeMath.sub(credit[msg.sender], amount)
```

5 Run-Time Monitoring to Control Execution Traces

Even though EVM execution is single-threaded, transactions are submitted in parallel, and miners may reorder and interleave those transactions arbitrarily [10, 32]. For this reason, it is possible to have data races as well as nondeterministic transaction orders. These are illustrated through two examples in the following.

[7] SafeMath transforms expressions in a binary manner. That is, for exception handling in compound expressions, say, for instance, a + b + c, SafeMath needs to be invoked compositionally.

5.1 Transaction Order Nondeterminism Issue

Figure 5, shows a simple Solidity contract GetterSetter containing two functions. `get` function allows a user to query the contract for the balance and the `set` function allows a user to update the balance with the value passed as an argument and return the old value of the balance. If this contract is concurrently executed by two customers, the answer would be non-deterministic. For instance, consider the following two scenarios:

Scenario 1:
1. C1 calls set(100); C1 calls get(); - returns 100
2. C2 calls set(50); C2 calls get(); - returns 50

Scenario 2:
1. C1 calls set(100); C2 calls set(50);
2. C1 calls get(); - returns 50
3. C2 calls get(); - returns 50 Nondeterminism depicted in (Scenario 2) is essentially due to interleaving of operations of transactions.

```
3  contract GetterSetter {
4      uint private balance;
5
6      function get() public returns(uint) {
7          return balance;
8      }
9
10     function set(uint x) public returns(uint) {
11         uint t = balance;
12         balance = x;
13         return t;
14     }
15 }
```

Fig. 5. Original getter setter contract

5.2 Overcoming Transaction Order Dependency

Here, the pattern for vulnerability is due to data races that could be avoided by taking into account the accessibility of the procedures of the contracts and also understanding the concurrent procedures that can co-exist without interfering with each other. The transformed body-part of Solidity is shown in Fig. 6 considering the access pattern: (GET SET); that is, `set` can be accessed by a process only after[8] a `get`. Let us assume several processes can call `get` in a concurrent

[8] This is a constraint we have enforced for simplicity; we could enforce controls like only one writer is permitted on a shared resource whereas multiple readers are allowed on that shared resource. This will become clear when we discuss nondeterminism in ERC20.

```
contract GetterSetter {
    uint private balance;
    mapping (address => string) public lastCall;

    function get() public returns(uint) {
        lastCall[msg.sender] = "get";
        return balance;
    }

    function set(uint x) public returns(uint) {
        require(keccak256(abi.encodePacked(lastCall[msg.sender]))
        ==keccak256(abi.encodePacked("get")),
        "set() should be called after get()");
        lastCall[msg.sender] = "set";
        uint t = balance;
        balance = x;
        return t;
    }
}
```

Fig. 6. Getter setter with access control

```
contract GetterSetter {
    uint private balance;
    address lastGetCalled;
    mapping (address => string) public lastCall;

    function get() public returns(uint) {
        require((lastGetCalled==address(0x0) || lastGetCalled==msg.sender),
        "concurrent execution by another contract!");
        lastGetCalled = msg.sender;
        lastCall[msg.sender] = "get";
        return balance;
    }

    function set(uint x) public returns(uint) {
        require(lastGetCalled==msg.sender,
        "concurrent execution by another contract!");
        require(keccak256(abi.encodePacked(lastCall[msg.sender]))
        ==keccak256(abi.encodePacked("get")),
        "set() should be called after get()");
        lastCall[msg.sender] = "set";
        uint t = balance;
        balance = x;
        lastGetCalled = address(0x0);
        return t;
    }
}
```

Fig. 7. Getter setter with concurrency & access control

fashion (note that multiple reading does not interfere with each other). Figure 6 and Fig. 7 show runtime checks (on the object-state) introduced in the body of Solidity shown in Fig. 5 so that access of actions is as per requirement.

5.3 ERC20 Token Standard and Issues of Nondeterminism

A large number of Ethereum development standards focus on token interfaces that help in ensuring the composable nature of smart contracts. ERC20 is the earliest token standard by Ethereum. Its brief interface[9] specification[10] is given below:

1. functions totalSupply(), balanceOf(address account), allowance () as well as emit events do not change the state.
2. `function transfer(address recipient, uint256 amount) external returns (bool)`: transfers a specified number of tokens from the caller to a different address (and reduces the balance of tokens by that quantity) if available. The `transfer` function is called to transfer tokens from the sender's account to a different one and it returns a boolean value, that is always true.
3. `function approve(address spender, uint256 amount) external returns (bool)`: It grants an allowance for another user. Note that there is no requirement that the sender should have at least the number of tokens granted at that time (or even later).
4. `function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);` i.e., `transferFrom` is used by the grantee to spend the allowance that has been approved earlier by some other user or owner for it.
   - Functionally, the operation (i) transfers an amount less than or equal to the allowance that has been approved by some other owner/user, (ii) and reduces the allowance by that amount, and (iii) the operation fails if the amount to be transferred is greater than the approved allowance (if that happens, it reverts).
   - It is this function along with `approve` that leads to possible misuse.

Remarks: The main differences between `transfer` and `transferFrom` are:

- In the case of `transfer`, the owner of the account grants certain number tokens to others. Thus, book keeping of the tokens can be done exclusively by the owner and there is no need of others to interfere. *Note that in an asynchronous system, the owner of a resource can control the order of his own transactions but not control the order of transaction of other users.*
- While in the case of `transferFrom`, the book keeping of the tokens cannot be done exclusively by the owner; the others (grantees by the owner) have to assist in book keeping as and when they use the partial or the full grant. Semantically, it can be interpreted as an account (or variable) that can be modified by multiple accounts/users; for a more detailed discussion the reader is referred to [33]. The issue with multiple users writing onto one account becomes clear in the sequel.

[9] https://ethereum.org/en/developers/docs/standards/tokens/erc-20/.
[10] https://ethereum.org/en/developers/tutorials/erc20-annotated-code/.

Attacks Scenario in ERC20. Consider a scenario of users (say owners of accounts) $\{U_1, U_2, \cdots, U_n\}$. Let A_{ij} $\forall i \neq j$ be the `Allowance` approved by U_i to users $U_j, \forall i \neq j$. It is to be noted users $\{U_1, U_2, \cdots, U_n\}$ are to be treated as asynchronous or loosely coupled processes. Possible attack scenario on ERC20 are well-documented[11]. A basic attack scenario is described below:

1. U_1 approves 80 tokens to U_2 (i.e., U_{12} is 80)
2. U_1 realises that he made mistake and he should have approved only 50 tokens.
3. To correct the mistake, U_1 sends out an approval using `Approve` for 50 tokens to U_2.
4. Now there are following two possibilities:
   (a) It is possible that U_2 has already withdrawn x tokens, $0 \leq x \leq 80$, using `transferFrom`. Thus U_2 will get additional 50 tokens, over and above x.
   (b) If U_2 is still to transfer, then the allowance U_{12} will get set to 50 tokens as wanted by U_1 perhaps (assuming he had done a mistake).

The basic reason for such an anomalous behaviour is that users $\{U_1, U_2, \cdots, U_n\}$ are asynchronous and hence lead to nondeterminism as the program does not use either a locks or a synchronization construct. A general programmer understands that the execution on blockchains like Ethereum using smart contracts is sequential and thus, does not realize the underlying nondeterminism that could arise due to interleaving of method calls. This is nicely illustrated in [22], where a following parallel is drawn:

> Accounts using smart contracts in a blockchain are like threads using concurrent objects in shared memory with correspondences of artifacts as below: *contract state to object state, call/send to context switching, Reentrancy to (Un)cooperative multitasking, Invariants to Atomicity and Nondeterminism to Data races.*

While one could have illustrated the attack scenario using only two users, we introduced n users to understand the general complexity. We can observe from the interface:

1. `transfer` can be interpreted as follows: The user say U_i transfers to some other account U_j, $j \neq i$. This corresponds to the owner sending his tokens to other accounts. Looking at the function in an isolated manner, it is like its storage is written only by himself (owner) and nobody else.
2. `transferFrom` can be interpreted as follows: Here, user U_i, takes $x \leq U_{ji}$ tokens (allowed by Owner U_j to U_i), and transfers to some other user U_k, $i, k \neq j$. Thus, U_i writes into the storage owned by U_j.
3. Looking at (1)–(2) above, we can see the storage of U_i is being written by the other processes (grantees of U_j) as well. If we treat each of the account of U_i as an asset, one can see that each account in principle can be written by other accounts/users who are not its owners. In other words, the abstraction of the

[11] https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/edit\#.

stakeholders can be seen as an n-shared asset problem discussed in [4,18] using a slight variation of ERC20 standard. In fact, the authors of [4] show that n-shared asset problem (hence ERC20) requires strong synchronization requiring consensus rather than just atomic registers and show that the bound on consensus number is dependant on the *object state* at that instant.

4. Overlaying the use of `approve` together with (1)–(3) provides a complete picture of execution of ERC20 usage in a given deployed context.

Thus, to establish the correctness of ERC20 deployment, we not only have to show that the attack scenario relative to `approve` is not possible but also show that the execution traces arising from (1)–(2) above also do not introduce any new attack scenarios.

To prevent such interleaving executions, the run-time monitor approach discussed in Sect. 5.2 can be adapted. Informally speaking, we need to assure, non-interfering execution of the various methods `approve(), transfer() and transferFrom()` instantiated from the users. This could be done through clauses ACCESS and PARALLEL discussed above or INVAR discussed in Sect. 6.3. It may be noted that realizing the clauses need to track the object state at that instant. For lack of space, we shall not go into details here; for further discussions, the reader is referred to [33].

6 Realizing Run-Time Monitors Through Annotations

In this section, we shall first discuss how the user specifies the annotations and indicate those that could be treated as standard, the basis of generating run-time monitors automatically from annotations and the scope of annotations in the context of smart contracts.

6.1 Annotations for Solidity Contracts and Standard Annotations

In the previous sections, we have illustrated various scenarios showing the exploitation of vulnerabilities like re-entrancy through the DAO, transaction order indeterminacy, nondeterminism in ERC20 tokens, and shown how run-time monitoring can be used to prevent exploitation of the underlying vulnerabilities. Now, a question arises:

> Is it the case that every time the programmer has to arrive at the run-time monitoring code? or Is there a general methodology to generate the run-time monitors automatically?

In the following we illustrate, as to how the programmer can specify the annotations.

6.2 Specification of Annotations

The programmer need not have to write the run-time monitoring code but just annotate the methods or expressions. With such annotations, the run-time

monitors can be automatically generated and integrated with the smart contracts. We shall illustrate typical annotations for the vulnerabilities discussed above and these will be treated as **standard annotations**.

A. *Annotation for Simple DAO - Attack1*: The main problem has been the re-entrancy in `withdraw`. The suggested annotation is:

NON-REENTRANT `withdraw`()

   – Interpretation: Prevent `withdraw` from re-entrancy at run-time.

B. *Annotation for Simple DAO - Attack2*: The main problem has been the re-entrancy of `withdraw` method as well as the overflow of the decrement expression. The annotation to be provided is:

NON-REENTRANT `withdraw`()
SafeMath (credit[msg.sender]-=amount)

   – while the first one is the same as above, the second one says the expression `credit[msg.sender]-=amount` should be run-time checked for arithmetic overflow and underflow.

C. *Annotation for Getter Setter Contract*: It has two annotations:
   1. `ACCESS (get), (get set)`
      – ACCESS annotation describes the execution order of calls of methods for each process (user). It essentially describes the trace of the possible method calls in the execution traces. In the annotation above, it is interpreted as a call to `get` or if `set` is called it has to be preceded by a `get`; in fact one could use regular expressions for specifying the access patterns [7] over method signatures. In a sense, the regular expression form could be: get* ∪ (get set)*. The interpretation is an user can call repeatedly `get` or follow the pattern of, first `get` and then `set` repeatedly.
   2. `PARALLEL (get get), (get set)`
      – The interpretation is: `get` can be called by several users possibly concurrently but at most one user can schedule `set` at any time subjected to the constraints specified in the ACCESS annotation.

Remarks: *A general question is: Why not realize "re-entrancy" or arithmetic exceptions for all the methods or expressions?* As Solidity execution requires gas, one need to monitor only where needed; one approach could be to first use it in general and debug the program through assertions to eliminate those that do not require run-time checks.

6.3 Generating Run-Time Monitors from Standard Annotations

From Sects. 4.2, 4.3, and 5.2, it is easy to see that the annotations as specified above can indeed generate the guards and the respective code-snippets for the checking. However, we shall illustrate below that the annotations can be treated

as declarations and then the run-time monitors are generated similar to the run-time checks introduced by the compilers in classical concurrent programming languages.

6.4 Declarations in a Shared Variable Programming Language

Our method of generating run-time monitors from standard annotations is based on the technique of *DECLARATIONS* used in concurrent programming languages to specify constraints on the execution of the program (used by the compiler to appropriately generate static/run-time checks as the case may be).

Programming languages for concurrent and distributed computing was one of key areas of research in 1980's [19]. The main rationale of specification (declaration) section of a concurrent program was meant to capture the interaction of processes [3,34] and the interaction of shared resources with processes. According to [3] concurrent programming languages have the following four important goals: expressiveness, data integrity, security and verifiability [3]. There have been a variety of concurrent programming languages keeping the discussed rationale. In early stages, languages were designed keeping in view the correctness -usually realized through discipline in usage either through formal specification or through formal declarations clauses. For instance, one of the early well designed languages, Ada, was quite disciplined from a concurrent perspective; here integrity was realized through mutual exclusive access of shared resources. For this reason, Ada rendezvous was considered inefficient as it did not permit concurrency even when the operations were non-interfering. There have been lot of research to realize efficiency/performance without foregoing correctness. For instance [34] explored a language structure to realize data integrity without unnecessary mutual exclusion. A typical shared variable program is a set of processes and shared resources and its structure is depicted in Fig. 8.

Clauses Import/Export highlight resources/services that can be imported or exported. The clause INVAR highlights the invariant property of the underlying resources. Clauses in the section CONSTRAINTS, vary based on the permitted interaction of processes and access of shared resources. For instance [34] requisite declarative clauses have been introduced so that data integrity can be realized without enforcing mutual exclusion unnecessarily and a methodology of establishing formal correctness through interference freedom proofs among processes and resources under the given constraints is described. The section 'Trans' highlights the operations that are possible on the resource. The MODBODY describes implementation of the operations. Further, the correctness of the program with classes ACCESS, or PARALLEL is established in [34].

Generating Run-Time Monitors

It is easy to see that the approach of `DECLARATIONS` highlighted above can be adapted for smart contracts by observing the similarity between the smart contract and the shared variable program. The authors of [22,32] have nicely demonstrated the similarity of smart contracts with that of a distributed programming structure through the following parallels: accounts using smart contracts in a

```
Process <process name ... >
Import ....          (* Classical syntax *)
Body
end Process

MODSPEC Shared < Resource name>
IMPORT < components being imported into the Module>
Export < components being expored outside the Module>
INVAR  < invariance on resource>
NONRENTRANT (...)  (* This is the additional clause for smart contracts *)
PARALLEL (...) (* Lists methods that could be instantiated concurrently *)
ACCESS (...)   (* Regular expressions on operations that provide the
                                                  trace structure *)
CONSTRAINTS  (*for interaction among processes and   resources
                                Is specified in varied forms  *)
Trans    (* various operations/functions/procedures that operate
               On the resource *)
Entry procedure p1 (...)
Entry procedure p2 (...)
...
Entry procedure p2 (...)
end ModSPEC
```

Fig. 8. General structure of a distributed program

blockchain are like threads using concurrent objects in shared memory, contract state with object state, call/send with context switching, re-entrancy with (un)-cooperative multi-tasking. Thus, now we can use the same technique with additional clause like `REENTRANT` for smart contracts; note that re-entrancy does not happen in classical languages as there is no fallback function.

For standard annotations discussed above, NON- REENTRANT, ACCESS, PARALLEL, by looking at the original Solidity contract and the Modified Solidity contract, it should become clear that based on the declarations and the information in the contract, the run-time monitors can be generated. We shall not discuss the algorithms here due to lack of space. We have built a pre-processor[12] for Solidity with *Annotations*, using the comment structure, that transforms it into a semantic preserving Solidity program satisfying by the declarations both statically and at run-time. The structure of the system is depicted in Fig. 9. The translation essentially uses the semantics of the respective annotation as an additional method of the contract and transforms the original program by guarded execution of the methods using the constructs like `require, revert` of Solidity as needed (we have also used `assert` statement as well for purpose of debugging). The method corresponding to the semantics of the annotation clause shall have the abstraction of events that are needed for guard expres-

[12] Snehal Borse, Prateek Patidar Solidity+: Specification Enhanced Solidity to Overcome Vulnerabilities, M.Tech. Dissertation, Department of Computer Science and Engineering, IIT Bombay 2019.

sions. Note that the structure of the transformed program remains unchanged, and hence, debugging the original program and the transformed program can be done with the same ease by the programmer; this is quite apparent from our earlier illustrations.

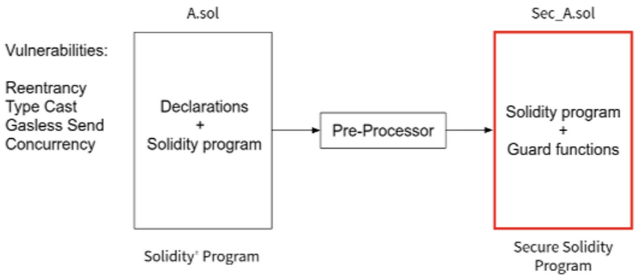

Fig. 9. Transformer for solidity program with declarations

Scope of the Approach

We have applied the approach to a large number of contracts that have vulnerabilities like re-entrancy, exceptions, type cast, gasless send, block.timestamp, Tx.origin, transaction order dependency etc., and successfully executed the contracts preventing the vulnerabilities being exploited.

Even though our approach is run-time oriented, we wanted to compare with various tools, even if they are static analysis based, to get a view of the utility and also the claim on the naturalness of debugging at the source level. For this purpose, we evaluated our system with Mythril[13] – a tool targeted to detect common vulnerabilities including integer underflow, owner-overwrite-to-Ether-withdrawal, etc., using symbolic executors (that may even be unsound often as the exploration space is limited) through bug injection in programs. We took a sample of 50 benchmark smart contracts [14] and found that our approach detects all those errors with succinct error messages apart from handling run-time errors as illustrated.

There are several programming tricks used by programmers to realize randomness or enforce time constraints, mismatch of interpretation of private/public for realizing secrecy. These errors have also been able to handle through guarded execution through the user annotations.

Another major vulnerability is that of *ether lost in transfer* - loss of *ether* happens when *ether* is sent to killed contracts or unknown addresses. This happens due to the underlying semantics of Solidity. One of the well known attacks exploiting such a vulnerability is the Parity fiasco (1-Parity and 2-Parity errors) [1]. As the general problem of detecting such losses is generally undecidable, the authors [27] have built a static analysis tool called, MAIAN, that

[13] https://mythril-classic.readthedocs.io/en/master/about.html.

classifies vulnerabilities into greedy, prodigal, suicidal, and posthumous statically. Our run-time framework could be used by annotating the *owners* in the contract and prevent a majority of such losses.

More Expressive Contracts. *Conditional Requirements*: Let us consider a conditional requirements like: *B can send Y to C only if it has received X from A*. It should be easy to see that the ACCESS class can be used to specify the appropriate annotation. This class of contracts are in general difficult to handle and often seen as anomalous behaviour of the blockchain execution.

General Enforcement of Constraints: Runtime verification approaches like specifying constraints using automata and timers have been explored in [11]. For instance, consider the first three clauses of the general English language contract taken from an interface description in [11].

1. The casino owner may deposit or withdraw money from the casino's bank, with the bank's balance never falling below zero.
2. As long as no game is in progress, the owner of the casino may make available a new game by tossing a coin and hiding its outcome. The owner must also set a participation cost of their choice for the game.
3. Clauses 1 and 2 are constrained in that as long as a game is in progress, the bank balance may never be less than the sum of the participation cost of the game and its win-out.

In [11], the programmer need to specify the requirements in a specification language LARVA that generates the needed run-time monitor. In our approach, noting that the constraint specified in (3), i.e., *as long as a game is in progress, bank balance may never be less than the sum of the participation cost of the game and its win-out* becomes the INVAR described in Fig. 8. Note that INVAR is an invariance of the contract and guarded execution of methods can be written using annotations like $at(method1) -> Cond1$, where $at(methdod1)$ denotes that the control point of the program at the beginning of $method1$; similarly $after(method1)$ denoted the control point after the completion of the body of method1. It can be easily seen, that programmer will find it easy to specify such annotations using INVAR that will be distributed over the methods by our preprocessor to make sure that the condition holds true always. For lack of space, we shall not go into further details.

Clause 3 of the legal contract says: "as long as a game is in progress, the bank balance may never be less than the sum of the participation cost of the game and its win-out." To ensure that Clause 3 holds we need to guard the function withdraw to maintain condition as an invariance (can be captured through the INVAR clause) as also other accessed functions.

7 A Comparative Discussion

We have explored a large number of smart contracts and instrumented them to check/validate their behaviour during execution. In particular, typical contracts

with vulnerabilities underlying standard annotations explored earlier like re-entrancy (DAO Attack 1 or Attack2, unknown caller), type cast, transaction order nondeterminism (ERC20, getter-setter), exceptions, etc., when executed are prevented from executing the contract further with appropriate exceptions and error messages to the user. Vulnerabilities such as unchecked send, gasless send are also being handled through annotations at the respective points. We have integrated our runtime monitor with a symbolic executor that provides a trace that would contain these assertions while passing through the remix debugger.

There is yet another set of vulnerabilities that has caught attention through the Parity Multisig attacks (Parity-1 and Parity-2) like characterizing prodigal, suicidal, greedy, posthumous contracts [27]. As already mentioned, a majority of the losses due to such vulnerabilities can be captured in our run-time framework using annotations about the owner, users who can change the ownerships etc. Such an enrichment is being experimented in our new prototype being engineered in Python by Hrishikesh Saloi, and Mohammad Ummair, at IIT Bombay.

Compared to transformational approaches [6,35], our approach works on full language without resorting to EVM level. Proof-theoretic approaches [2,10,13, 16,20,21,29] that work on Solidity cum EVM have their own subtleties, complexities and cannot be easily adapted by an ordinary programmer. Thus, usability of our approach with integration to the Remix debugger has distinct advantages in building trusted smart contracts. Static analysis tools such as [25,27] are quite useful but have the limitations of static analysis of a dynamic language like Solidity. In fact, just like the undecidability of the access control problem (often called Harrison, Ruzzo and Ullman (HRU)), it can be shown that using the `delegatecall` etc., the "rights" of contracts will remain undecidable (we shall not go into details here). Hence, while static analysis approaches are useful in developing trusted programs as well as postmortem analysis, run time monitoring is preferable in our view where run-time checks cannot be avoided as we also need to keep in view the wallet/user smart contracts. As compared to systems like [24], ours is general and keeps to the debugging of a given contract rather than a general guideline and run time monitors are generated automatically.

Runtime verification approaches like [11], require specifications in exclusive specification languages like LARVA. Whereas our approach needs simple annotations from which one can generate the semantic preserving original contract keeping the annotations as constraints. This enables easy debugging of the contract and further provides a rough proof-outline gaining trust on the contract.

In summary, our run-time framework can be effectively used to overcome a spectrum of vulnerabilities through standard annotations and general annotations, as the run-time checks monitor the conditions on object state and enforce the synchronization requirements of smart contracts (cf. [4]).

8 Conclusions

In this paper, we have proposed a run-time framework for preventing some of the major vulnerabilities through simple intuitive programming annotations in Solidity contract through standard annotations from which the required run-time monitors can be generated automatically. It is also shown how the annotations described are similar in nature to that of declarative clauses in concurrent programming languages; the declarative clauses have been serving concerns of correctness as well as expressive power. Further, we have shown how the scope of the approach is quite expressive to include conditional contracts, contracts with overall constraints etc. Some of the main characteristics of our approach are:

1. For a large class of vulnerabilities, the run time monitors can be constructed automatically with standard annotations.
2. The approach of annotations is quite expressive. The INVAR clause can be effectively used for expressing a quite a lot of constraints as well as for constructing formal proofs; this class can also express some of the constraints expressed through ACCESS and PARALLEL.
3. The similarity of annotations with that declarative clauses also makes it possible to use those annotations in a modular way.
4. The transformation works at the Solidity level that preserves the original program structure and hence, debugging becomes natural.
5. As the run time monitors are on the blockchain, programmer gains confidence or trust on the contract even relative to his wallet contract and the run-time checks overcome false positives that are prevalent in static analysis methods and provides a rough proof-outline of the execution of the contract.

While we have illustrated similarity of standard annotations and declarations, we are working towards formal specification of the additional "declarative" structure needed in Solidity with a formal semantics. While the approach is definitely useful and extensible to different contexts, it would be nice to: (1) adapt the approach for proof carrying smart contracts [10] as that would really enhance the trust on the smart contract usage, (2) explore algorithms for elimination of run-time checks, and (3) optimize run-time checks to optimize the use of gas. It would be nice to have a constructive comparative evaluation of the various smart contract languages highlighted earlier to benefit smart contract programmers both qualitatively and quantitatively.

Acknowledgments. The work was carried out at the Centre of Excellence for Blockchains supported by Ripple Corporation. I wish to thank Snehal Borse, and Prateek Patidar who initially were responsible for building the basic prototype and to Hrishikesh Saloi and Mohammad Ummair, who have been re-engineering the system in Python to handle a spectrum vulnerabilities using this approach. Special thanks to Dr. Vishwas Patil of Center for Blockchain Research for his constructive comments on an initial draft of the paper.

References

1. Akentiev, A.: Parity multisig github. https://github.com/paritytech/parity/issues/6995
2. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, pp. 66–77. ACM, New York (2018). https://doi.org/10.1145/3167084
3. Andrews, G.R., McGraw, J.R.: Language features for process interaction. SIGOPS Oper. Syst. Rev. **11**(2), 114–127 (1977). https://doi.org/10.1145/390018.808318
4. Androulaki, E., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, pp. 30:1–30:15 (2018)
5. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
6. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS 2016, pp. 91–96. ACM, New York (2016). https://doi.org/10.1145/2993600.2993611
7. Campbell, R.H., Habermann, A.N.: The specification of process synchronization by path expressions. In: Gelenbe, E., Kaiser, C. (eds.) OS 1974. LNCS, vol. 16, pp. 89–102. Springer, Heidelberg (1974). https://doi.org/10.1007/BFb0029355
8. Crafa, S., Di Pirro, M., Zucca, E.: Is solidity solid enough? In: Bracciali, A., Clark, J., Pintore, F., Rønne, P.B., Sala, M. (eds.) FC 2019. LNCS, vol. 11599, pp. 138–153. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-43725-1_11
9. Das, A., Balzer, S., Hoffmann, J., Pfenning, F.: Resource-aware session types for digital contracts. CoRR abs/1902.06056 (2019)
10. Dickerson, T., Gazzillo, P., Herlihy, M., Saraph, V., Koskinen, E.: Proof-carrying smart contracts. In: Zohar, A., et al. (eds.) FC 2018. LNCS, vol. 10958, pp. 325–338. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-662-58820-8_22
11. Ellul, J., Pace, G.J.: Runtime verification of Ethereum smart contracts. In: 2018 14th European Dependable Computing Conference (EDCC), pp. 158–163 (2018). https://doi.org/10.1109/EDCC.2018.00036
12. Ethereum: Solidity documentation (2018). https://solidity.readthedocs.io/
13. Filliâtre, J.-C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
14. Ghaleb, A., Pattabiraman, K.: How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection. In: ISSTA 2020. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3395363.3397385
15. Grishchenko, I., Maffei, M., Schneidewind, C.: Foundations and tools for the static analysis of Ethereum smart contracts. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 51–78. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_4
16. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of Ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) POST 2018. LNCS, vol. 10804, pp. 243–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_10

17. Grossman, S., et al.: Online detection of effectively callback free objects with applications to smart contracts. **2**(POPL) (2017)
18. Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovič, M., Seredinschi, D.A.: The consensus number of a cryptocurrency. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, pp. 307–316. Association for Computing Machinery, New York (2019)
19. Hansen, P.B., Dijkstra, E.W., Hoare, C.A.R.: The Origins of Concurrent Programming: From Semaphores to Remote Procedure Calls. Springer, Heidelberg (2002)
20. Hildenbrandt, E., et al.: KEVM: a complete formal semantics of the Ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF), pp. 204–217 (2018)
21. Hirai, Y.: Defining the Ethereum virtual machine for interactive theorem provers. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 520–535. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_33
22. Kolluri, A., Nikolic, I., Sergey, I., Hobor, A., Saxena, P.: Exploiting the laws of order in smart contracts. CoRR abs/1810.11605 (2018)
23. Krishna Rao, M.R.K., Kapur, D., Shyamasundar, R.K.: Proving termination of GHC programs. New Gen. Comput. **15**(3), 293–338 (1997). https://doi.org/10.1007/BF03037949
24. Lee, J.H.: DappGuard: active monitoring and defense for solidity smart contracts (2017)
25. 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, CCS 2016, pp. 254–269. ACM, New York (2016). https://doi.org/10.1145/2976749.2978309
26. Mueller(ConsenSys), B.: Mythril: a classic security analysis tool for Ethereum smart contracts (2017). https://github.com/ConsenSys/mythril
27. Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, pp. 653–663. Association for Computing Machinery, New York (2018)
28. Openzeppelin contributors: Safemath.sol (2019). https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol. Accessed 16 June 2019
29. 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. 414–430. Los Alamitos, CA, USA (2020)
30. Remix: Remix - Solidity IDE (2018). https://remix.ethereum.org/
31. Schneidewind, C., Grishchenko, I., Scherer, M., Maffei, M.: Ethor: practical and provably sound static analysis of Ethereum smart contracts (2020). https://doi.org/10.48550/ARXIV.2005.06227
32. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 478–493. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_30
33. Shyamasundar, K.R.: A safety assessment of token standards for Ethereum: Erc20 and beyond. In: 6th Symposium on Distributed Ledger Technology (SDLT). SDLT, Gold Coast, Australia (2022)
34. Shyamsundar, R.K., Thatcher, J.W.: Language constructs for specifying concurrency in CDL. IEEE Trans. Softw. Eng. **15**(8), 977–993 (1989). https://doi.org/10.1109/32.31354

35. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.: Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, pp. 67–82. ACM, New York (2018). https://doi.org/10.1145/3243734.3243780
36. Zhang, W., Banescu, S., Pasos, L., Stewart, S., Ganesh, V.: MPRO: combining static and symbolic analysis for scalable testing of smart contract. arXiv abs/1911.00570 (2019)