Oracle-Supported Dynamic Exploit Generation for Smart Contracts

Haijun Wang, Ye Liu, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu

Abstract—Despite the high stakes involved in smart contracts, they are often developed in an undisciplined manner, leaving the security and reliability of blockchain transactions at risk. In this paper, we introduce ContraMaster—an oracle-supported dynamic exploit generation framework for smart contracts. Existing approaches mutate only single transactions; ContraMaster exceeds these by mutating the transaction sequences. ContraMaster uses data-flow, control-flow, and the dynamic contract state to guide its mutations. It then monitors the executions of target contract programs, and validates the results against a general-purpose semantic test oracle to discover vulnerabilities. Being a dynamic technique, it guarantees that each discovered vulnerability is a violation of the test oracle and is able to generate the attack script to exploit this vulnerability. In contrast to rule-based approaches, ContraMaster has not shown any false positives, and it easily generalizes to unknown types of vulnerabilities (e.g., logic errors). We evaluate ContraMaster on 218 vulnerable smart contracts. The experimental results confirm its practical applicability and advantages over the state-of-the-art techniques, and also reveal three new types of attacks.

Index Terms—Smart contract, test oracle, security vulnerability, fuzzing.

1 INTRODUCTION

SMART contracts are computer programs that execute on top of blockchains (e.g., Bitcoin [1] and Ethereum [2]) to manage the flow of funds, exchange of assets, and transfer of digital rights between various parties [3], [4]. Transactions through smart contracts are stored persistently on the blockchain and thus immutable, without requiring a central third party to validate them. Due to these unique advantages, smart contracts have gained a lot of popularity and attention in recent years. Many believe that this technology has the potential to reshape a number of industries, e.g., banking, insurance, supply chains, and financial exchanges [5].

The role of smart contracts in managing shared assets (often cryptographic currencies) requires a high level of security and reliability. Yet, an increasing number of high-profile attacks have occurred, resulting in great financial losses. Such attacks are facilitated by the lack of a rigorous development and testing process. One notorious example is the "DAO" attack, where attackers stole more than 3.5 million `Ether` (equivalent to about $\$45$ million *USD* at that time) from "DAO" contract [6].

These incidents have spurred activities in detecting vulnerabilities in smart contracts [6], [7], [8], [9], [10], [11], [12]. Existing techniques usually detect smart contract vulnerabilities based on rule-based approaches: the contract behaviors are matched to a limited set of *vulnerability patterns* identified beforehand. The precision and recall of these techniques largely depend on the size and quality of their collections of vulnerability patterns. Most such patterns are defined at the syntax level such as particular statements/calls sequences, ignoring their actual effects on contracts and resulting in false positives. For example, Zeus [11] treats the `refund` function in the DaoChallenge [13] contract (shown in Fig. 9) as vulnerable, because it may potentially be reentered. However, such reentrancy behavior cannot be exploited in stealing `Ether` from it, since the authors have incorporated defensive mechanisms to prevent from transferring unauthorized `Ether` (c.f. Sect. 5.6.2). Similar problems exist in how ContractFuzzer [8] detects *exception disorder* (more details in Sect. 5.6.2).

To address above issues, we propose to dynamically execute transactions and observe their actual effects on the contract states in order to detect exploitable vulnerabilities. Our key insight is that almost all the existing (syntactical) vulnerability patterns result in a (semantic) mismatch between the externally visible events (e. g., amount transferred and contract balance changed) and the internal contract states (e. g., amount maintained in the contract's internal bookkeeping) in a transaction. As a result, the internal bookkeeping becomes inconsistent, indicating a successful exploit. Based on this observation, we define a general-purpose semantic test oracle, which can be used to detect such mismatches at runtime.

Our technique is dynamic and works on target contracts that run on a realistic test environment. Thus, it does not suffer from the imprecision faced by most static techniques. All the vulnerabilities detected by our approach can be successfully reproduced. When generating attack inputs, we take into account the unique characteristics of smart contracts which make traditional fuzzers ineffective. For example, attackers need to synthesize a sequence of transactions to successfully mount an attack (e.g., the transaction sequence "*deposit* $\rightarrow$ *withdraw*" is required for the DAO attack) [14]. In contrast, traditional fuzzers such as AFL [15] focus on vulnerabilities triggered by a *single* test case. We extend traditional grey-box fuzzers with mutation operators customized

H. Wang was with the School of Computer Science and Engineering, Nanyang Technological University, Singapore.
Y. Liu, Y. Li, S. Lin, and Y. Liu are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore.
L. Ma is with the Faculty of Information Science and Electrical Engineering, Kyushu University, Japan.
C. Artho is with the School of Computer Science and Communication, KTH Royal Institute of Technology, Sweden.

for smart contracts, including *transaction sequences*, *gas limits*, *fallback functions*, and *contract states*, apart from *function inputs*. We also develop the novel feedback mechanisms to guide the fuzzing process, by considering the data-flow and dynamic contract state information, together with the control-flow information.

We implemented our approach in ContraMaster and evaluated it on 218 vulnerable smart contracts reported by ContractFuzzer [8] and Zeus [11]. We found that, of these potentially vulnerable contracts, only 28 (12.84 %) are exploitable, and the remaining (87.16 %) are not. In addition, ContraMaster detected 26 hitherto unknown vulnerabilities, which could not be detected using previously identified vulnerability patterns.

In this paper, we make the following novel contributions:

- We design a general-purpose semantic oracle, which can be used to detect a wide range of vulnerabilities, such as reentrancy, exception disorder, gasless send and integer overflow/underflow.
- We develop an oracle-supported dynamic exploit generation framework for smart contracts—ContraMaster. Specifically, we design customized mutation operators and feedback mechanisms, which are shown useful at improving the effectiveness of vulnerability detection.
- We evaluate ContraMaster on 218 smart contracts and demonstrate its advantages in discovering *exploitable* vulnerabilities over state-of-the-art techniques. Among the 218 vulnerabilities reported by the state-of-the-art techniques, only 28 are exploitable, and ContraMaster detects all of them without false positives.
- We present our findings on the 26 newly identified vulnerabilities, which cannot be detected by previously identified vulnerability patterns.

The rest of this paper is organized as follows. Section 2 provides the necessary background and definitions for the rest of the paper. Section 3 illustrates our semantic test oracle. Section 4 introduces the technical details of our oracle-supported fuzzing and automated exploit generation. Section 5 discusses challenges for implementing ContraMaster and the evaluation results on real Ethereum smart contracts. Finally, we discuss related work and conclude in Sects. 6 and 7, respectively.

2 Preliminaries

In this section, we provide the necessary background and definitions for the rest of the paper.

2.1 Blockchain and Smart Contract

A *blockchain* is a shared, transparent distributed ledger, and is maintained by a decentralized network of peers (miners) [16]. The miners perform the mining process of adding a block and verifying the validity of transactions through a *proof-of-work* (PoW) [17] or other consensus protocols, such as *proof-of-stake* (PoS) [18]. Thus, a blockchain can be considered as an ever-growing list of blocks, each encoding a sequence of transactions, always available for inspection and safe from tampering. Each block contains a cryptographic signature of its previous block. No previous block can be changed or rejected, unless 51% of miners are controlled and all its successors are changed or rejected. With this structure, blockchain achieves decentralization, traceability, transparency, and immutability.

A *smart contract* is computer program which allows users to define and execute transactions automatically on the blockchain [19]. A smart contract resides at a specific address on the blockchain, providing a number of publicly accessible functions and fields. Moreover, a special *balance* variable records the cryptocurrencies owned by the contract address and cannot be freely altered by programmers. When a function of the smart contract is invoked, the current state of the contract is retrieved from the blockchain, and the updated state of the contract is stored back on the blockchain after execution.

A transaction is carried out in the form of a message sent to a particular address on the blockchain, which can either be a normal user account address or a contract address. A user sends transactions to the blockchain in order to: (1) create new contracts, (2) invoke a function of a contract, or (3) transfer cryptocurrencies to contracts or other users. All the transactions sent by participants, called *external transactions*, are recorded on the blockchain. Upon receiving an external transaction, a contract can also trigger some internal transactions, which are not explicitly recorded on the blockchain, but still have effects on the balance of participants or contracts.

The Ethereum Virtual Machine (EVM) [2] is a stack-machine with an instruction set including standard arithmetic instructions, conditional and unconditional jump instructions, basic cryptography primitives, and primitives for gas computation. The data is stored on the persistent memory area storage (a key-value store that maps 256-bit words to 256-bit words), the contract-local memory (a contract obtains a freshly cleared instance for each message call), or a stack (since the EVM is not a register machine but a stack machine, all computations are performed on the stack). When Ethereum smart contracts are compiled and deployed, they are run on the EVM.

Gas is a unit used to measure the amount of computational effort taken to execute certain operations on EVM, e.g., storage and read. The fees paid to miners to execute an operation can be calculated by the amount of gas multiplied by the gas prices. Higher fees would attract more miners into the system, and make the system as profitable and alluring as possible for miners. Besides, they can prevent malicious adversaries from launching DoS attacks, since each transaction incurs a cost. However, if the provided gas is not enough, the transaction execution would be terminated and exception is thrown once gas is burnt out. If we do not handle such exception correctly, it may introduce vulnerabilities (e.g., gasless send, c.f. Sect. 3.3).

Fallback function is an unnamed function in Ethereum smart contracts. This function is externally visible and does not have any argument or return anything. It is executed on a call to the contract if none of the other functions match the given function identifier [20]. Specially, if the contract receive the plain Ether (without data supplied), the fallback function would be executed. In the worst case, its implementation can only rely on 2,300 gas being available. For example, in order to avoid reentrancy (c.f. Sect. 3.3) when the `send` or `transfer` function is used to transfer Ether, only 2,300 gas

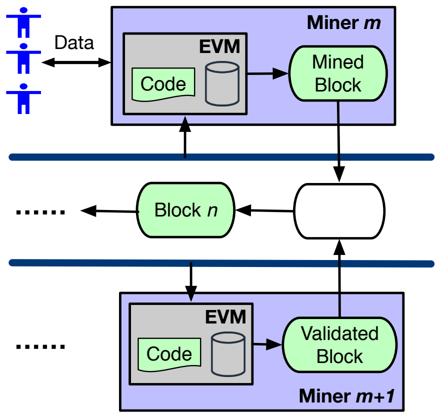

Fig. 1: Illustration of Ethereum smart contracts running on blockchains.

is pre-allocated for the fallback function so that it cannot call another function.

Figure 1 is a schematic representation of the Ethereum smart contracts running on blockchains. When participants submit transactions to the blockchain, they may be mined by a *miner* m and executed on m's EVM. When the transactions are finished, the transactions and their execution results are sealed in a new block appended to the block list and propagated over the blockchain network. At this point, other miners, such as a miner $m+1$, may discover the new block and validate it. When majority of the miners validate and agree on the new block, it becomes the confirmed block, and cannot be tampered.

2.2 A Customized Semantic Model

Smart contracts are similar to general computer programs in the sense that they are written in a Turing-complete language, e. g., Solidity [21]. Various formal semantic models of Solidity have been proposed in previous work [11], [22], to enable formal verification of smart contracts. Since our goal in this paper is to perform dynamic analysis for Solidity contracts and verify runtime contract states against our test oracle, we propose a customized semantic model that is simple but expressive enough to capture the *transaction-related* contract behaviors. We define *contract state* and *transaction* as follows.

Definition 1 (Contract and Contract State)**.** *A* contract *can be abstracted as a tuple* $C := \langle c, bal, A, \sigma \rangle$*, where* $c \in Addr$ *is a unique address identifying the contract,* $bal \in \mathbb{N}$ *is the externally visible* balance *of the contract,* $A \in 2^{Addr}$ *is a set of* account addresses *of participants, and* σ *is the internal contract state. A* contract state σ *is a type-consistent valuation of the global variables* (V)*.*

The set of all states is denoted by $\Sigma \cup \{Err\}$, where Err is a special state indicating an *error state*. For a given state $\sigma \in \Sigma$ and an expression e, e_σ denotes the evaluation of e in that state. The semantics of a contract program is a set of *execution traces*, where a trace corresponds to a sequence of internal contract states. In Ethereum, each contract also has an externally-visible *contract balance* (*bal*) representing the total amount of funds in the contract, which is a part of the *blockchain state* (as opposed to the contract state).

Definition 2 (Transaction)**.** *A* transaction $t := \langle s, r, v \rangle$*, if performed successfully, deducts amount* v *from the sending account's balance (*$s.bal$ *where* $s \in Addr$*) and transfers the funds to a receiving account at address* $r \in Addr$*. We denote the values of a variable* g *before and after the transaction as* $pre(g)$ *and* $post(g)$*, respectively.*

A transaction usually alters the blockchain state, reflected as updates on the contract balance. In this case, the caller and callee contracts of a transaction correspond to the sending and receiving accounts, respectively.

2.3 Threat Model

To study the potential vulnerabilities of smart contracts, an important guarantee of a securely implemented contract is that it only allows *authorized* accounts to transfer the *authorized* amount of `Ether` [23]. For example, the DAO contract, if implemented correctly, should only allow users to withdraw the amount of `Ether` have been deposited previously. We assume that a regular user with no special capabilities attempts to break through this guarantee. A smart contract is *vulnerable*, if adversaries can bypass authorization and steal more `Ether` from the contract than allowed, resulting in a loss for the contract owner, or victims store `Ether` in the contract but receive a lower level of authorization than intended, resulting in a loss for the participants [23].

Besides, vulnerabilities may happen due to the inter-play of multiple smart contracts. We do not limit the number of contract files under test, whether they are (1) compiled and deployed at the same address, (2) deployed at different addresses and communicate through delegate calls, or (3) deployed at different addresses and communicate through transfer messages. For (2) and (3), we consider attacks which involve only a two-way communication. In other words, when there are a group of contracts interacting with each other, the attacker in our attack model picks one of them as the direct target to communicate with. This is enough for modeling most of the commonly seen vulnerability types from the literature.

3 Semantic Test Oracle

The fundamental difficulty in detecting smart contract vulnerabilities is the lack of a general-purpose test oracle. This is because smart contracts do not crash like general computer programs, and their execution may be silently reverted in cases of irregularities. To address this issue, we propose a test oracle which detects irregularities in smart contracts at the semantic-level. Our semantic test oracle implements two types of invariants that transactions must comply with.

3.1 Balance and Transaction Invariants

Smart contracts are mainly used to manage the transfer of assets and perform bookkeeping [24], thus they need to keep track of participants' individual account balances, called the *bookkeeping balances*. Contract programs use an internal *bookkeeping variable* (e. g., `balances` in Fig. 2) to record the bookkeeping balances. Suppose a bookkeeping variable $m : Addr \mapsto \mathbb{N}$ is given.

Definition 3 (Balance Invariant)**.** *For every contract* $\langle c, bal, A, \sigma \rangle$, $\sum_{a \in A} m_\sigma(a) - bal = K$, *where* K *is a constant.*

The *balance invariant* requires that the difference between the contract balance and the sum of all participants' bookkeeping balances remains constant, before and after a transaction. This invariant is defined within a single contract, i. e., *intra-contract*, and it ensures the integrity of the bookkeeping balances. If the bookkeeping balances are not updated correctly within a transaction, then the violation of this invariant indicates that an irregular event has happened. For example, when an integer underflow happens during a transaction, the contract balance naturally goes down while the bookkeeping balances go up instead.

Definition 4 (Transaction Invariant)**.** *For every outgoing transaction* $\langle c, r, v \rangle$ *where* c *is the sending contract's address,* $\Delta(m_\sigma(r)) + \Delta(r.bal) = 0$, *where* $\Delta(x) = \mathrm{post}(x) - \mathrm{pre}(x)$.

The *transaction invariant* requires that the amount deducted from a contract's bookkeeping balances is always deposited into the recipient's balance. This *inter-contract* invariant ensures the consistency between the both ends of a transaction. Note that the consistency of incoming transactions can be guaranteed by the balance invariant or other contract's outgoing transaction invariant. In some cases, a transaction in progress may fail and funds are not transferred. If the failure is not be captured by the contract's bookkeeping variables, the contract may become vulnerable (e. g., exception disorder).

3.2 Runtime Invariant Checking

Since the invariants are supposed to hold for each transaction among contracts involved, the test oracle can be implemented as a set of runtime checks before and after each transaction. Notice that the runtime checks mentioned here are on the transaction-level, in which multiple contracts are involved. Thus, adding assertions in contracts does not work since assertions in each contract cannot express inter-contract properties. The biggest challenge of implementing such runtime checks is to automatically identify the bookkeeping variables.

Identification of Bookkeeping Variables. Most contracts performing meaningful transactions among multiple participants contain a bookkeeping variable, usually with the name `balances` or `balanceOf`. The bookkeeping variable has a few characteristics distinguishing it from the others: (1) it is a mapping from account addresses to unsigned integers, i.e., `mapping(address => uint*)` (there are a few exceptions which are explained in Sect. 5); (2) it is at least updated once in a `payable` function; and (3) in a normal transaction, the amount received from an account address should be reflected as a balance increase for that address.

Based on these observations, we design an algorithm for the automatic identification of bookkeeping variables. For every mapping variable updated in `payable` functions, we send several transactions with randomly chosen values (including extremely large and small amounts). We then observe the increased amount at the sender's address. If the increases always match with the amounts being sent, we record the variable as a bookkeeping variable.

```
1  contract SimpleDAO {
2    mapping (address => unit) public balances;
3    function donate(address to) {
4      balances[to] += msg.value;
5    }
6    function withdraw(uint amount) {
7      require(balances[msg.sender] >= amount);
8      msg.sender.call.value(amount)();  //Exception Disorder
9      balances[msg.sender] -= amount;
10   }
11   function another_withdraw(uint amount) {
12     require(balances[msg.sender] >= amount);
13     msg.sender.send(amount)();  //Gasless Send
14     balances[msg.sender] -= amount;
15   }
16 }
```

Fig. 2: A simple contract susceptible to the "DAO" attack.

```
1  contract attackDAO {
2    SimpleDAO public dao;
3    constructor(address _dao){
4      dao = SimpleDAO(_dao);
5    }
6    function donate(address to){
7      dao.donate.value(msg.value)(to);
8    }
9    function withdraw(uint amount){
10     dao.withdraw(amount);
11   }
12   ......
13   function () public payable {
14     dao.withdraw(...); //Reentrancy
15   }
16 }
```

Fig. 3: Reentrancy attack on the simple "DAO" contract.

The bookkeeping variables in some contracts may not refer to the amount of `Ether`. This is often the case in ERC-20 and ERC-721 contracts [25]. In these contracts, participants' digital assets are reflected in terms of the number of available *tokens* rather than `Ether`. In such cases, standard APIs for getting individual account balances (`balanceOf`) and total contract balances (`totalSupply`) in terms of tokens are provided and can be directly used to implement the runtime checks.

3.3 Detecting Vulnerabilities with the Test Oracle

Now we discuss how previously reported vulnerabilities [7], [10], [16], [26] can be detected by our test oracle.

Reentrancy. Programmers often believe that, when a non-recursive function is invoked, it cannot be re-entered before its termination. However, this is not always the case, due to the fallback function introduced by Solidity. Take the simplified "DAO" attack for example. Two contracts, `SimpleDAO` (the victim, in Fig. 2) and `attackDAO` (in Fig. 3), are deployed on the blockchain. The reentrancy vulnerability of `SimpleDAO` can be exploited by `attackDAO`. When `attackDAO` withdraws from `SimpleDAO` via Line 10 of Fig. 3, it will execute Lines 7–8 of Fig. 2. Then, due to fallback function mechanism, Line 8 of Fig. 2 executes Line 14 of Fig. 3, which further executes Lines 7–8 of Fig. 2 again and thus generate recursive calls. Notice that, the execution of Line 9 of Fig. 2 is delayed.

The consequence of reentrancy is that Line 9 of Fig. 2 may be executed more times than allowed, and it leads to the integer underflow of bookkeeping variable `balances`. The underflow will produce the incorrect values for `balances`, which violates the balance invariant (Definition 3).

```
contract another_attackDAO {
  ......
  function () public payable {
    throw; //No reentrancy
  }
}
```

Fig. 4: Exception disorder example.

```
1 contract UnderflowAttack {
2   ...
3   function withdraw(uint amount) public {
4     require(balances[msg.sender] - amount > 0);
5     msg.sender.transfer(amount);
6     balances[msg.sender] -= amount; //Underflow
7   }
8   ...
9 }
```

Fig. 5: The underflow attack example [27].

Exception Disorder. Solidity is not uniform in handling exceptions. Within a chain of nested calls, there are two types of exception handling mechanisms [26]: (1) If a function in the chain is a `call` (the same for `delegatecall` and `send`), the exception is propagated along the chain, reverting all side effects, until it reaches the `call`. From that point on, the execution is resumed with the `call` returning `false`. (2) If all the functions in the chain are direct calls (not via `call`, `delegatecall` and `send`), the execution stops and all side effects are reverted, including the transfers of `Ether`.

Developers may handle exceptions incorrectly. For example, Line 8 of Fig. 2 tries to transfer `Ether` to account `msg.sender`. If this account is a contract, this transfer may fail, resulting in an exception. Since this exception is not properly handled, the balance of this account (`balances[msg.sender]`) is decreased but it does not actually receive `Ether`. Thus, this transaction will violate the transaction invariant (Definition 4).

Gasless Send. *Gasless send* is a special case of exception disorder. When transferring `Ether` from one contract to another with function `send`, it may lead to an out-of-gas exception. The default gas limit for function `send` is 2,300 `Wei`. If the recipient's fallback function contains too many instructions, it may lead to an out-of-gas exception for function `send`, resulting in a *gasless send*. For example, at Line 13 of Fig. 2 the attacker (whose address is `msg.sender`) may have an expensive fallback function and the `send` function may fail. Since *gasless send* is a special case of exception disorder, it can also be detected by the transaction invariant (Definition 4).

Integer Over/Under-flow. Smart contracts heavily use integer arithmetic operations to manipulate participants' balances. However, these variables are susceptible to integer over/under-flow, e.g., in Fig. 5, `balances[msg.sender]` and `amount` are both unsigned integers. If `balances[msg.sender]` is less than `amount`, the check at Line 4 will pass due to integer underflow, leading to another underflow at Line 6. This produces the wrong value for the bookkeeping variable, which violates the balance invariant (Definition 3).

Other Vulnerabilities. In this work, we focus on contracts used for managing funds of multiple participants and handling monetary transactions among them, which constitute the majority of the Ethereum smart contracts. For example, our proposed balance and transaction invariants apply for all standard token-based contracts, e.g., ERC-20 and ERC-721, as well as many other contracts with the identification of the bookkeeping variables.

There are a few other types of vulnerabilities, including the *timestamp dependency*, *block number dependency*, and *freezing ether* [26]. Exploiting vulnerabilities such as timestamp and block number dependencies requires the cooperation of miners, therefore cannot be easily realized at the contract-level. In model checking [28] terms, freezing ether is an violation of the *liveness property*, while our dynamic approach can only detect violations of *safety properties*.

It is possible to extend the test oracles to handle more types of contracts and vulnerabilities. For example, invariants on proportional token distribution [29], i.e., the values of two exchanged tokens are proportional, have been proposed to cover a more general case. Our fuzzing framework (see Sect. 4) is independent from the test oracles and thus can also be to extended with other user-defined invariants to detect more types of vulnerabilities.

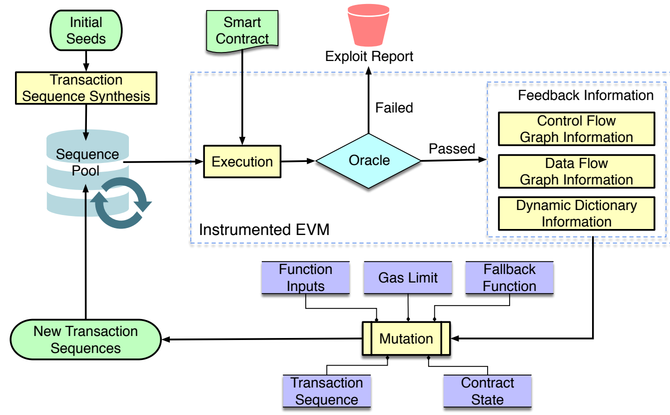

Fig. 6: Overview of the ContraMaster.

4 Oracle-Supported Fuzzing

Figure 6 shows the overview of ContraMaster, which is driven by a grey-box fuzzing loop [15], [30]. Given a set of initial seeds, ContraMaster randomly synthesizes a set of transaction sequences and picks one from the pool in each iteration of the fuzzing loop. ContraMaster runs each transaction in sequence on an instrumented EVM. When the transaction is finished, ContraMaster verifies the contract state against our *semantic test oracle* (details in Sect. 3). If a violation is detected, ContraMaster reports a vulnerability and presents the generated attack contract and transaction sequence which can be used to reproduce the exploit. Otherwise, ContraMaster collects runtime execution information to guide the test sequence generation for the next iteration. The information collected mainly includes control-, data-flow graph and contract state information.

Furthermore, ContraMaster is equipped with a number of new mutation operators customized for smart contracts (c.f. Sect. 4.1). In addition to *function inputs* used in traditional fuzzers, we also use *gas limits*, *fallback functions*, *transaction* *sequences* and *contract states* as the mutation targets. Many vulnerabilities in smart contracts require the interplay of several contracts and can only be exploited by a particular transaction sequence with the correct gas limit [31]. Therefore, the customized mutation operators are important for triggering vulnerabilities. The newly generated inputs are added to the pool and the fuzzing process continues until it exceeds the allocated resource limits.

4.1 Oracle-Supported Fuzzing Algorithm

The goal of the fuzzing component is to automatically generate transaction sequences that violate the test oracle. Algorithm 1 presents its high-level idea. Given a contract program C and a set of initial seeds T, it first generates a set of initial transaction sequences TS from T (Line 1). In every iteration of the fuzzing loop (Lines 2 to 22), we select a transaction sequence ts from TS (Line 3), and initializes the current execution trace sequence (E) and contract state dictionary ($dict$) as $empty$ (Line 4). ContraMaster then executes each transaction $t_i \in ts$ (Lines 5 to 10), and collects its execution trace e_i (Line 6). It would also stores observed contract states (e.g., the values of bookkeeping variables and contract balances) into dictionary (Line 7), similar to what is done in traditional fuzzers such as VUzzer [32]. The dictionary values are later used in generating function inputs. The execution trace of each transaction e_i is concatenated to form a transaction sequence trace E (Line 8). After that, ContraMaster checks whether the test oracle is violated. If so, it adds the current transaction sequence ts into the output TS', which is a script exploiting the vulnerability (Line 10).

ContraMaster performs mutations at both the level of single transaction (Lines 11 to 18), and at the level of transaction sequences (Lines 19 to 22). If t_i achieves new branch coverage, we perform mutations on the function inputs and gas limit (Lines 13 to 14). When fallback function is called in the transaction, we also perform mutations on fallback functions at Line 17. If the transaction sequence's trace E has new data dependence coverage, we perform transaction sequence mutation (e.g., switching the order of transactions) at Line 20. In the end, we randomly reset the whole smart contract states by contract state mutation at Line 22. More details about each mutation strategy are given in Sect. 4.2.

The highlighted code in Algorithm 1 shows the differences of ContraMaster from traditional grey-box fuzzing approaches such as AFL [15]. To summarize, traditional fuzzing techniques work on a single call, while ContraMaster works on a transaction sequence. The reason is that a lot of vulnerabilities can only be triggered by a sequence of transactions. To effectively generate such sequences, we use data-flow information to guide its mutation, which cannot be achieved by control-flow information. Another important difference is that ContraMaster performs mutations on fallback functions, through which the attack contracts may interact with the target contract.

4.2 Mutation Strategies

In this section, we present our five mutation strategies, namely, the mutation of *function inputs*, *gas limit*, *fallback function*, *transaction sequence*, and *contract state*.

Algorithm 1: Oracle-Supported Fuzzing

input : a contract program C, a set of initial seeds T
output: transaction sequences TS' violating the test oracle

1 $TS \leftarrow$ generate initial transaction sequences from T ;
2 **while** *time budget not reached and abort signal not received* **do**
3 $\quad ts \leftarrow selectNext(TS)$;
4 $\quad E, dict \leftarrow \emptyset, \emptyset$;
5 $\quad$ **foreach** *transaction* t_i *in* ts **do**
6 $\quad\quad$ run t_i and collect execution trace e_i on EVM ;
7 $\quad\quad$ $dict \leftarrow extractValues(e_i)$ // dictionary values ;
8 $\quad\quad$ $E \leftarrow E, e_i$ (append e_i to E) ;
9 $\quad\quad$ **if** *test oracle is violated* **then**
10 $\quad\quad\quad TS' \leftarrow TS' \cup ts$;
11 $\quad$ **foreach** *transaction* t_i *in* ts **do**
12 $\quad\quad$ **if** t_i *has new branch coverage in* e_i **then**
13 $\quad\quad\quad TS_i \leftarrow$ InputMutate(t_i, $dict$) ;
14 $\quad\quad\quad TS_g \leftarrow$ GasMutate(t_i) ;
15 $\quad\quad\quad TS \leftarrow TS \cup TS_i \cup TS_g$;
16 $\quad\quad\quad$ **if** t_i *executes fallback function* **then**
17 $\quad\quad\quad\quad TS_f \leftarrow$ FallbackMutate(t_i) ;
18 $\quad\quad\quad\quad TS \leftarrow TS \cup TS_f$;
19 $\quad$ **if** E *has new data dependence coverage* **then**
20 $\quad\quad TS_t \leftarrow$ TransSeqenceMutate($ts, E, dict$) ;
21 $\quad\quad TS \leftarrow TS \cup TS_t$;
22 $\quad$ ContractStateMutate() ;

Attack Contract. ContraMaster uses the attack contracts to interact with target contract, thus we first automatically generate the attack contract, like in the example shown in Fig. 3. To synthesize the attack contracts, we use a variable to represent the target contract and initialize it in the constructor function (Lines 2 to 5). Then, for each function in target contract, ContraMaster develops a surrogate function to call this function, as shown in Lines 5 to 11 in Fig. 3. Finally, we synthesize the fallback function as shown in Lines 12 to 14.

Function Inputs. Line 13 of Algorithm 1 mutates the parameters passed to each target function. We consider two types of function parameters: primitive and array types.

Primitive-type parameters include Booleans (`bool`), account addresses, unsigned integers (`uint*`), integers (`int*`), and arbitrary-length raw byte data (`byte*`). First, we pick special values from the dynamic dictionary of previously seen state variable values with matching types to generate multiple *mutation ranges*. Within these ranges, we randomly generate values as candidate function inputs. Second, we opportunistically negate bits in these inputs to produce new inputs (similar to the "flip1" operation used in AFL [15]). For account addresses, we simply enumerate addresses from a predefined account list. In most cases, the collected dynamic dictionary and "flip1" are enough for generating effective inputs, since most smart contracts have relatively simple program logic.

For array types, we consider both fixed- and arbitrary-length arrays. For fixed-length arrays of primitive-type elements (e. g., `address[n]` and `uint*[n]`), we use the same technique described above to generate random values for each element. For an arbitrary-length array, we first generate a positive random number as the array length, and then proceed as dealing with a fixed-length array. For arbitrary-length bytes or strings, we use values from the dictionary and mutate them with bit flips.

Gas Limit. Every instruction executed on EVM has an associated fee, known as the *gas*. If the gas cost of a transaction exceeds the *gas limit*, an out-of-gas exception is thrown. To simulate all possible behaviors with the exceptions thrown in the middle of a transaction, we mutate on the gas limits at Line 14 of Algorithm 1. First, we estimate the maximum gas cost G_t and the intrinsic gas cost G_i [33] (consisting of a constant transaction fee and a data-dependent fee) for a target transaction. Then, we divide the range between G_i and G_t into n equal intervals and randomly choose a gas limit from each interval to initiate the transaction with.

Fallback Function. The fallback function is an important mechanism in Ethereum and is highly relevant to the reentrancy and exception disorder vulnerabilities. When receiving funds from the target contract under test, the attacker's contract may use specially crafted fallback function to perform malicious activities. To trigger these behaviors, ContraMaster performs mutations on the fallback function at Line 17 of Algorithm 1.

We generate multiple attack contracts with different fallback functions to interact with the target contract, such as in Figs. 3 and 4. In particular, we allow any function of the target contract to be called within the attacker's fallback function. Apart from that, we also have an empty fallback function and one that contains a single `throw` statement (e.g., `revert`()) to trigger exception disorders.

Transaction Sequence. Some vulnerabilities can only be triggered with the correct transaction sequences. For example, the DAO attack can only be mounted by first depositing into the target contract and then withdrawing from it. To find a successful exploit, we mutate the call sequences as follows. For a given candidate transaction sequence, (1) we randomly insert/remove transactions to/from it; and (2) if two transactions in a sequence operate on the same contract state variable, we switch their order.

Contract State. The effects of a transaction depend on the contract state in which it is initiated. To mutate contract states, we allow the values of state variables to be carried forward across multiple test runs and reset the state periodically, say, after every n transactions. The intuition behind resetting a contract to restore its initial state is that some contract states are not useful for discovering new vulnerabilities. For example, when a DoS attack is launched on the "King of the Ether Throne" contract [34], no further state can be reached. In this case, resetting the contract state helps to escape from dead-ends and find more vulnerabilities. The reset of contract state is achieved by redeploying the contract code to the private test network, which does not pose significant overhead.

4.3 Feedback Mechanisms

The feedback used by ContraMaster can be broadly categorized into the *control-driven* and *data-driven*, and *contract state* feedback information. The control-driven feedback mechanism strives to cover more CFG edges as with AFL [15], by favoring uncovered CFG edges.

Data-Driven Feedback. Since smart contract is state-relevant, we should synthesize a suitable transaction sequence to detect the vulnerabilities. However, the transaction sequence cannot be guided by the control-flow information, as a different transaction sequence does not necessarily cover new CFG edges. Thus, we propose to use data flow to guide transaction sequence mutations. If the mutated transaction sequence covers new data dependencies, it is an interesting transaction sequence. We first define the data dependency as follows.

Definition 5 (Data Dependency [35], [36])**.** *There is a data dependency from y to x if there exists a directed path p from x to y where x defines a variable $v \in V$, y uses v and there is no node $z \in p$ that redefines v.*

In the execution of transaction sequence, if two transactions operate on the same contract state variable, we switch their order to generate new transaction sequence. For example, the transaction sequence "*withdraw* $\rightarrow$ *deposit*" both operate on the bookkeeping variable *balances*, thus we switch their order to generate a new transaction sequence "*deposit* $\rightarrow$ *withdraw*", which may trigger the reentrancy vulnerability.

Contract State Feedback. Apart from the data dependency, we also use the contract states (Definition 1) to guide the function input generation. The basic idea is that, in most cases, the execution of current transaction heavily depends on the contracts' states. For example, the sequence "*deposit* $\rightarrow$ *withdraw*" may trigger the reentrancy, but it depends on whether the funds deposited is greater than the funds withdrawn. Thus, we use contract states to guide function input generation, such that the less funds withdrawn than those deposited. to subsequently trigger potential vulnerabilities. In fact, we extract the dynamic contract states as a dynamic dictionary, which is similar to VUzzer [32]. However, the latter uses the `immediate` values in the code as the static dictionary.

4.4 ContraMaster by Example

Taking the DAO contract (Fig. 2) as an example, we illustrate the workflow of ContraMaster with one possible set of generated transaction sequences shown in Fig. 7. Based on the initial seeds, i.e., $\{$*withdraw(10)*, *deposit.value(5)(*)*$\}$, ContraMaster randomly generates a transaction sequence, "ts_1 = *withdraw(10)* $\rightarrow$ *deposit.value(5)(*)*", shown on the first row of Fig. 7. Here we assume that the attack contract used is *another_attackDAO* from Fig. 4. After ts_1 is executed, we identify a data dependency between the `withdraw` and `deposit` functions over the state variable `balances` by analyzing the data flow of the execution trace. Based on the *data-driven feedback*, we adopt the *transaction sequence* mutation strategy on ts_1 to generate a new one: "$ts_2 = deposit.value(5)(*) \rightarrow withdraw(10)$", as shown on the second row.

Yet, executing ts_2 does not expose a reentrancy issue, because the value to withdraw is more than the balance deposited earlier, thus resulting in a runtime failure. Here, ContraMaster inspects the contract states and adds "(`balances`, 5)" (the deposited amount) into the dynamic

No.	Transaction Sequences	Attack Contracts	Feedback Mechanisms	Mutation Strategies
1	$ts_1 = withdraw(10) \rightarrow deposit.value(5)(*)$	*another_attackDAO*	data-driven	transaction sequence
2	$ts_2 = deposit.value(5)(*) \rightarrow withdraw(10)$	*another_attackDAO*	contract state	contract state
3	$ts_3 = deposit.value(5)(*) \rightarrow withdraw(3)$	*another_attackDAO*	control-driven	fallback function
4	$ts_3 = deposit.value(5)(*) \rightarrow withdraw(3)$	*attackDAO*	—	—

Fig. 7: Illustration of the discovery of the DAO attack by ContraMaster.

dictionary. Using *contract state* as feedback, ContraMaster generates three groups of input parameters for `withdraw`, i.e., ≤ 5, 5, and ≥ 5. For example, we may end up with a transaction sequence, "$ts_3 = deposit.value(5)(*) \rightarrow withdraw(3)$", based on the *contract state mutation* strategy, as shown on the third row of Fig. 7.

To trigger a reentrancy attack, a suitable fallback function is still required in the attack contract. The fallback function in *another_attackDAO* (Fig. 4) does not reenter the `withdraw` function. Based on the *control-driven* feedback, ContraMaster mutates the *fallback function*, and may generate a new one such as the one in `attackDAO` (Fig. 3). At this point, we have successfully found an attack contract (i.e., *attackDAO* in Fig. 3) as well as a transaction sequence (i.e., "*deposit.value(5)(*)* $\rightarrow$ *withdraw(3)*"), which trigger reentrancy.

When reentrancy occurs, `Ether` would be transferred to the attacker's account via the `withdraw` function. In the end, the update of *balances* at Line 9 would underflow the `balances`. Since the value of the bookkeeping variable `balances` becomes inconsistent, the balance invariant (Definition 3) is violated.

5 Implementation and Evaluation

In this section, we present the implementation of ContraMaster and evaluate it on a set of benchmarks.

5.1 Implementation

We use a C++ implementation of the Ethereum client, Aleth v1.8.0, to setup a single node private blockchain as the test network, and Truffle Suite v4.1.14 as the test harness. ContraMaster consists of a front end which generates inputs and triggers transactions, and a back end which executes transactions on smart contracts and validates their behaviors.

The front end starts from some initial seeds and performs mutations to persistently generate new seeds based on our feedback mechanisms. In the back end, the Aleth EVM is modified to monitor the runtime execution: the test oracle is enforced by asserting invariants after each transaction is finished. We use big numbers (i.e., the `BigInt` javascript library) when handling values, arithmetic operations, and checking the invariants. Therefore, no overflow/underflow is possible in the analyzer. If an invariant violation is detected, the test sequence is reported as an exploit. Otherwise, it performs data-flow, control-flow, and contract-state analysis to provide feedback to the front end, which continues to generate new test inputs. In total, ContraMaster is implemented with more than 5,000 lines of Javascript, Python and C++ languages.[1]

1. The tool implementation and benchmark used in our experiments are available at: https://github.com/ntu-SRSLab/vultron.

5.2 Evaluation

Our empirical evaluation of ContraMaster tries to answer the following research questions:

- **RQ1:** How does ContraMaster perform compared to the state-of-the-art pattern-based approaches?
- **RQ2:** How effective is the feedback-guided test generation in speeding up the exploit generation?
- **RQ3:** How do the initial seeds affect the effectiveness of ContraMaster?
- **RQ4:** Can ContraMaster discover previously unknown vulnerabilities?

Setup. All our experiments were performed on a 64-bit Ubuntu 18.04 desktop with an Intel Xeon CPU E5-1650 (3.60 GHz, 12 cores) and 16 GB of RAM. Since we focus on smart contract vulnerabilities, not the consensus protocol, we configure only one peer node for the mining process. We set the initial mining difficulty of the genesis block to 1 so that transaction confirmation is fast. We also assume that each participant owns as much `Ether` as the total `Ether` supply at the time of writing (currently about 10^8 `Ether`).

Subjects. To evaluate our approach, we selected the experimental subjects as follows. We compared ContraMaster with the dynamic fuzzing tool ContractFuzzer [8] and the static verification tool Zeus [11]. These two tools reflect the state of the art in dynamic and static smart-contract analysis, and use properties that are stronger than our test oracle. Thus, we use the reported vulnerabilities as our experimental subjects (except *timestamp dependency*, *block number dependency*, and *dangerous delegatecall*, which cannot be easily exploited).

We included all the 188 contracts reported as vulnerable by ContractFuzzer into our benchmark. Since ContractFuzzer does not analyze integer overflow/underflow [8], we augment the set of benchmarks with 30 contracts containing overflow/underflow vulnerabilities. These 30 contracts were randomly sampled from a set of 1,095 smart contracts that reported by Zeus [11] to have this vulnerability. To evaluate the feasibility and ability of ContraMaster in identifying the bookkeeping variables, we first conducted an experiment on the 1,095 smart contracts used by Zeus. Out of the 514 smart contracts whose source code is available on Etherscan, our automatic identification method successfully found the internal bookkeeping variables in 430 smart contracts (about 83.6 %). We manually checked the bookkeeping variables found and confirmed these results to be correct. We have also manually investigated the contracts in which bookkeeping variables cannot be automatically identified and found that they either do not perform bookkeeping or use complicated data structures or user-defined types to record the bookkeeping. In total, we selected 218 smart contracts for our subsequent experiments.

TABLE 1: Vulnerabilities reported by ContractFuzzer, Zeus and ContraMaster.

Vulnerability Types	Pattern-based Detection #Vul	#Exp	(%)	ContraMaster
Reentrancy	14	6	42.86 %	6
Exception Disorder	36	13	36.11 %	13
Gasless Send	138	6	4.34 %	6
Integer Over/Under-flow	30	3	10.00 %	3
New Vulnerabilities	—	—	—	26
Total/Avg.	218	28	12.84 %	54

5.3 Comparison with the State-of-the-Art

RQ1 and RQ4 relate to the effectiveness of ContraMaster. To evaluate this, we compared ContraMaster with ContractFuzzer and Zeus. Table 1 shows the vulnerability type, the number of reported vulnerabilities, the number of actually exploitable vulnerabilities, and the percentage of exploitable vulnerabilities over the vulnerabilities reported by the state of the art. The last column lists the exploitable vulnerabilities reported by ContraMaster. Row "*New Vulnerabilities*" shows the new vulnerabilities ContraMaster found; these are different from the vulnerability types covered by the state-of-the-art ContractFuzzer and Zeus.

As is shown in Table 1, ContractFuzzer reports 14 reentrancy vulnerabilities. Out of these, only six contracts were reported exploitable by ContraMaster. For the eight smart contracts not reported by ContraMaster, we manually checked the contract code and confirmed that they are non-exploitable. Our investigation showed that ContractFuzzer over-reports reentrancy vulnerabilities (around 42.86 % exploitable) because its oracle is defined at the syntactic level.

For exception disorders, ContractFuzzer reported 36 vulnerabilities, while ContraMaster only reported 13. ContractFuzzer only considers a transaction being safe if the exceptional case is followed by a `throw` statement. However, an exception can be handled by multiple ways, e. g., reverting the modified variables, in which the exception would not lead to an exploitable vulnerability.

ContractFuzzer also reports 138 gasless send vulnerabilities. However, the gasless send vulnerabilities are not exploitable if the `transfer()` function is used to send `Ether`, because the `transfer()` function automatically reverts the program state if there is not enough gas. These cases were reported by ContractFuzzer as vulnerable. Out of the 138 gasless send vulnerabilities, only six were reported exploitable by ContraMaster (4.34 %).

Overall, ContraMaster has shown its advantages against ContractFuzzer in producing fewer false positives and the capability in finding new vulnerabilities. We did not compare the time taken for each case individually, because this information is not available for ContractFuzzer [8]. To evaluate the efficiency of ContraMaster, we constructed a baseline version, called ContraAFL, and further evaluated the efficiency of our mutation strategies (c.f. Sect. 5.4).

Integer overflow/underflows constitute another important issue in smart contracts. Zeus detects integer overflow/underflow based on the predefined syntactic patterns [37]. However, whether these really happen depends on the execution environment and program contexts. In the 30 sampled integer overflow/underflow vulnerabilities, only 3 were reported exploitable by ContraMaster (10.00 %).

Summary. In the 218 detected vulnerabilities by ContractFuzzer and Zeus, only 28 vulnerabilities were reported exploitable by ContraMaster (12.84 %). For the non-exploitable vulnerabilities, we manually checked and confirmed that they are indeed not exploitable. Furthermore, ContraMaster finds 26 new vulnerabilities, which are different from the vulnerability types in ContractFuzzer and Zeus.

From the above experiments, we observe that ContraMaster only reports exploitable vulnerabilities because its oracle is defined at the semantic level. In addition to that, ContraMaster is able to discover previously unknown vulnerabilities. Two of the authors have independently verified the results by replaying the exploit scripts manually. The authors of ContractFuzzer also confirmed our findings. Later, in Sect. 5.6, we explain why some vulnerabilities reported by ContractFuzzer and Zeus are not exploitable, and illustrate the new attacks.

5.4 Evaluation of Feedback Effectiveness

RQ2 questions the effectiveness of feedback in fuzzing. To answer this question, we implemented a variant of ContraMaster (called ContraAFL), which strips the data-dependence guidance from ContraMaster, similar to AFL. Then, we performed the experiments on the 23 exploitable examples (five exploitable vulnerabilities are repetitive in *exception disorder* and *gasless send*) by repeating each experiment eight times, and the compared the performance of ContraMaster and ContraAFL. We set a timeout of 120 seconds for each benchmark program, in which ContraMaster can successfully finish all experiments.

Figure 8 shows the comparison results, where the x and y axes show the time taken by ContraMaster and ContraAFL, respectively. ContraMaster performs better than ContraAFL for points above the diagonal line, which was observed for all examples we ran. From the results, we can see that ContraMaster is highly efficient, compared with ContraAFL, in recognizing exploitable vulnerabilities. Specially, three exploitable vulnerabilities cannot be found by ContraAFL in the given timeout (points lying on the top x-axis) in their all experiments.

Furthermore, a manual investigation revealed that the test sequences generated by ContraAFL are mostly meaningless. For example, ContraAFL often chooses amounts of `Ether` to send that are larger than the amount it owns. Thus, the transaction would be reverted, resulting in no actual effect on the smart contracts. On the other hand, ContraMaster is guided by feedback and gradually generates meaningful transactions.

Mann Whitney U-test Scoring. Following Klees et al.'s [38] recommendation, we apply the *Mann Whitney U-test* on the time used to find the vulnerabilities. As shown in Table 2, in most experiments (16 out of 23 cases), the p-values are smaller than or close to a significance level of 0.05. Thus, we conclude there exists a statistically significant difference in the time used to find the vulnerabilities, compared to ContraAFL.

Vargha and Delaney $\widehat{A}_{12}$ Scoring. To determine the extent to which ContraMaster outperforms ContraAFL, we also use

TABLE 2: Statistical results for feedback-directed fuzzing.

Contracts	ContraMaster (s)		ContraAFL (s)		Statistics		Seeding Strategies: $(seqLen, funParam)$ (s)					
	Avg	Var.	Avg	Var.	p-value	$\hat{A_{12}}$	(1,0)	(2,0)	(all, 0)	(1, rand)	(2, rand)	(all, rand)
BountyHunt	8.37	63.94	46.04	2,073.61	0.015	0.828	76.43	106.86	—	81.71	56.14	1.24
CreditDepositBank	27.60	442.15	66.41	1,931.76	0.051	0.750	69.23	77.64	—	71.25	37.33	14.28
Etheramid	1.80	2.04	3.22	4.18	0.103	0.715	77.50	105.56	—	65.72	34.61	1.38
EthSplit	1.97	2.29	2.69	5.47	0.016	0.828	79.43	—	—	11.62	18.81	0.99
Eth_VAULT	3.33	20.64	75.21	2,153.68	0.006	0.906	79.08	—	—	14.18	15.29	16.31
FreeEth	82.70	2,327.13	100.46	1,156.77	0.203	0.609	—	—	—	91.92	108.27	48.71
HelpMeSave	12.25	60.20	26.41	286.13	0.063	0.734	107.96	—	—	94.58	72.79	27.18
HFConditionalTransfer	1.32	0.48	1.77	0.48	0.085	0.710	80.38	72.89	—	37.62	15.06	1.24
Honey	72.48	2,352.50	82.51	1,861.25	0.026	0.594	—	—	—	—	109.16	105.10
MultipicatorX4	7.77	32.45	18.10	81.32	0.009	0.859	—	—	—	80.68	23.82	16.24
MyToken	0.31	0.03	25.01	979.31	0.003	0.914	2.72	0.24	0.41	0.39	0.29	0.36
Pie	99.91	1,211.07	99.84	1,078.11	0.374	0.547	103.03	108.80	—	—	103.71	105.51
PIGGY_BANK	5.57	33.76	55.63	2,576.46	0.026	0.797	54.87	78.69	—	54.55	49.45	4.65
Private_accumulationfund	2.85	12.05	71.91	2,023.33	0.015	0.725	110.31	111.40	—	29.39	47.79	1.91
Private_Bank	48.30	2,053.90	105.95	1,015.69	0.021	0.797	—	108.71	—	112.62	—	80.50
PrivateDeposit	21.02	128.52	—	0.00	0.0002	1.000	—	96.29	—	—	66.29	23.73
SafeConditionalHFTransfer	2.14	2.81	3.61	7.91	0.264	0.602	54.25	86.09	—	26.81	18.17	1.53
SimpleCoinFlipGame	1.95	3.48	—	0.00	0.0002	1.000	43.17	—	—	16.88	8.21	1.43
SimpleLotto	106.41	213.88	111.25	132.40	0.025	0.601	—	—	—	16.6	1.06	1.28
Soleau	26.30	54.92	28.20	89.25	0.437	0.531	—	—	—	44.94	23.73	69.56
TokenBank	16.75	220.67	—	0.00	0.0002	1.000	—	—	—	—	65.73	8.10
transferIntwopart	1.68	1.52	3.10	3.50	0.103	0.725	85.65	—	—	8.32	10.16	16.07
WhaleGiveaway	87.09	1,825.72	93.94	1,376.56	0.368	0.555	—	110.32	—	90.22	94.05	86.69
avg	27.82	481.14	58.50	829.16			91.48	103.63	114.80	62.17	47.82	27.57

"—" indicates timeout (exceeding 120 seconds).

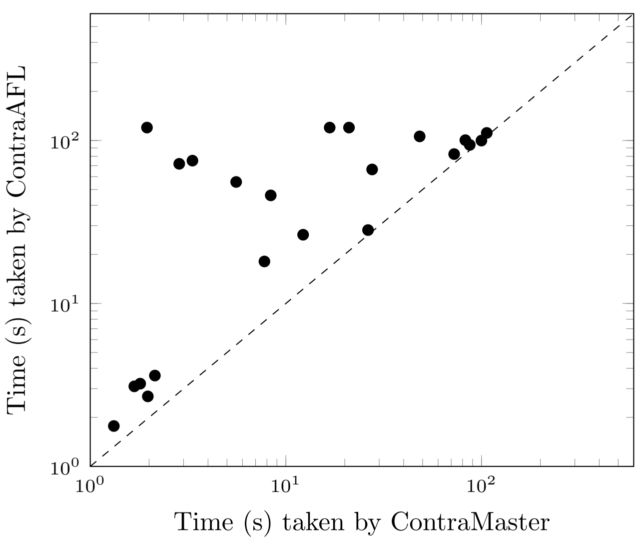

Fig. 8: Time taken by ContraMaster and ContraAFL.

Vargha and Delaney's $\widehat{A}_{12}$ statistical test [38], [39]. From Table 2, we can see among benchmark experiments the resulting $\widehat{A}_{12}$ statistic exceeds the conventionally large effect size of 0.71 in 16 out of 23 cases. Therefore, we conclude that the time usage in ContraMaster to find vulnerabilities is statistically different from that in ContraAFL.

Besides, we also developed another variant of ContraMaster, in which all our mutation strategies (e.g., contract states and data dependence) are disabled, and conducted the experiments again. The results show that this fuzzing technique only finds one vulnerability (i.e., from BountyHunt), and results in timeouts for all other subjects.

5.5 Evaluation of Initial Seeds

It is well established [38] that initial seeds may have great influence on the effectiveness of fuzzing techniques, since good initial seeds tend to trigger behaviors closer to the ones exposing vulnerabilities and thus require less mutations. We conducted a set of experiments to study the impact of initial seeds on the effectiveness of ContraMaster in finding vulnerabilities. We compared seeds of various length and different function inputs. In terms of lengths, we picked transaction sequences of length one, two, and up to the total number of functions available in the contracts. In terms of function inputs, we tested both all-zero input parameters and randomly generated ones. We combined these two seeding strategies and have generated six different types of initial seeds.

The experimental results are shown on the right side of Table 2. The columns "$(seqLen, funParam)$" list the average time taken to discover vulnerabilities when using initial seeds of lengths one ("1"), two ("2"), or the number of functions available ("all"), and function input parameters of either all zeros ("0") or randomly generated values ("rand").

Generally speaking, the randomly generated function inputs perform better than the all-zero inputs. The average time taken to find vulnerabilities are at least 1.5X longer and there are a lot more timeouts, when given all-zero inputs to the initial seeds. This is mainly because in contracts handling monetary transactions, all-zero function inputs usually do not generate meaningful contract state changes. For instance, if the function inputs contain a parameter representing the amount of transferred `Ether`, a zero input would trigger a revert of the transaction, not changing the state of the blockchain.

When it comes to transaction sequences, the results become more interesting. Specifically, with all-zero inputs, the longer the transaction sequences, the less effective they are in finding vulnerabilities. The inverse becomes true when randomly generated inputs are used, i.e., longer transaction sequences are more effective seeds. This suggests that, when all-zero inputs are used, increased length of transaction sequences lead to a waste of time on less effective seeds.

Yet, when utilizing randomly generated inputs, which have higher chance of triggering more diverse contract states, the increased length of transaction sequences would raise the chance of covering vulnerable contract states. From these results, we observe that initial seeds do have an impact on the effectiveness of ContraMaster and the use of longer as well as more diverse inputs is more preferable.

5.6 Case Studies

In this section, we report on interesting findings from our case studies.[2] Section 5.6.1 introduces some new attacks found by ContraMaster in the experiments, and Sect. 5.6.2 investigates on some non-exploitable vulnerabilities reported by ContractFuzzer [8], Zeus [11], and Oyente [9].

5.6.1 New Attack Surfaces

There are three different types of new attacks found by ContraMaster in 26 smart contracts.

Incorrect Access Control. Access control is important in smart contracts, which only allows critical operations to be performed by the owner of the contract. Access control-related issues are ranked the second most severe among all vulnerability types [40]. When access control is exploitable to steal ether, one of our oracle invariants (c.f. Definitions 3 and 4) will be violated. Thus, our approach is able to detect this vulnerability.

Honey Trap. Some contracts, e. g, `ETH_VAULT` and `WhaleGiveaway`, contain honey traps where the participants deposit `Ether` into the contracts, and cannot withdraw it again. These smart contracts are unfair to the participants. Our approach is able to detect this vulnerability because the behavior of honey traps violates the transaction invariant (c.f. Definition 4) of our test oracle.

Deposit Less and Withdraw More. Some smart contracts, e. g., `BountyHunt`, `LZLCoin` and `PowerCoin`, are vulnerable by allowing an adversary to withdraw more `Ether` than they have deposited, which violates the balance invariant (c.f. Definition 3) of our test oracle. Thus, our approach is able to detect this vulnerability.

5.6.2 Non-Exploitable Vulnerabilities Reported by the Pattern-Based Approaches

In this section, we illustrate the reasons why some vulnerabilities detected by existing techniques are not exploitable.

Reentrancy. Existing techniques, e.g., Zeus, can detect reentrancy based on predefined properties. However, these properties are fixed patterns and are too strong so that non-exploitable renentrancy is also reported. For example, the contract `DaoChallenge` shown in Fig. 9 was reported by Zeus as vulnerable [11]. However, based on the official website [41] where the contract was originally from, this is not an exploitable reentrancy.

```
1  contract DaoChallenge {
2    function withdrawEtherOrThrow(uint256 amount) private {
3      bool result = msg.sender.call.value(amount)();
4      if (!result) {
5        throw;
6      }
7    }
8    ...
9    function refund() noEther {
10     address sender = msg.sender;
11     uint256 tokenBalance = tokenBalanceOf[sender];
12     if (tokenBalance == 0) { throw; }
13     tokenBalanceOf[sender] = 0;
14     withdrawEtherOrThrow(tokenBalance * tokenPrice);
15     notifyRefundToken(tokenBalance, sender);
16   }
17 }
```

Fig. 9: An example for non-exploitable reentrancy.

```
1  contract Store {
2    function payout() returns (uint) {
3      uint amount = ownerBalances[msg.sender];
4      ownerBalances[msg.sender] = 0;
5      if (msg.sender.send(amount)) {
6        return amount;
7      } else {
8        ownerBalances[msg.sender] = amount;
9        return 0;
10     }
11   }
12   ...
13 }
```

Fig. 10: Non-exploitable exception disorder.

This contract first checks whether the balance of the message sender is zero at Line 11. If so, it throws an exception and reverts the program state. Otherwise, it sets the balance of the sender to zero at Line 12, and then uses `withdrawEtherOrThrow()`, the safe withdraw function, to fetch `Ether` at Line 13. Through this safe withdraw function, the program may re-enter the function `refund()`. When reentering function `refund()`, the balance will be set to zero. Thus, the reentrancy cannot pass the check at Line 11 again, and the program state is reverted. As a result, an adversary cannot steal `Ether` from this contract.

Exception Disorder. ContractFuzzer uses a pattern to detect exception disorders: it checks if a `throw` statement is executed after a failed `send()`, in order to revert the transaction [8]. Zeus [11] checks whether there is a write operation on a global variable after a failed `send()`. However, both checks are purely syntactic, and many non-exploitable vulnerabilities are reported because of this.

For example, consider the code snippet from `Store` in Fig. 10. At Line 5, the contract pays out `Ether` to the message sender and the `send()` operation may fail. When the `send()` operation fails, the contract reverts the program states at Line 8. This is in fact a correct way to handle the exception. However, it is reported as a vulnerability by both ContractFuzzer and Zeus. There is no easy way to precisely detect exception disorder without semantic understandings.

Integer Overflow/Underflow. Zeus used 1,523 smart contracts for evaluation, and reported 1,095 contracts from that dataset as susceptible to integer overflow/underflow [37]. Although many integer overflow/underflows may occur in theory, not all of them are practical. First, the total amount of `Ether` available on the Ethereum platform is limited to 140 million `Ether` [42]. Therefore, one cannot use infinite amount of `Ether` to overflow/underflow an

2. More details can be found at: https://sites.google.com/view/contramaster.

`uint256` variable. Second, the numbers of transactions and participants are also much smaller than the upper bound of `uint256`. Finally, many smart contracts use safe mathematics operations protected from overflow/underflow. This is not recognized by the property(pattern)-based techniques, such as Zeus.

5.7 Threats to validity

We have selected a benchmark set that is large enough to show the capabilities of ContraMaster and compare it with other tools. However, both the set of benchmarks we selected ours from, as well as our own selection, may include a certain sample bias. Thus, the results may not generalize to all smart contracts. Moreover, while our approach is generic and also applicable to other types of smart contracts, some implementation details and issues found are specific to Ethereum.

6 Related Work

Existing work on smart contract vulnerability detection can be categorized into static analysis [6], [7], [9], [10], [11], [43] and dynamic analysis [8], [12].

6.1 Static Analysis

Program Analysis. Securify [7] first infers semantic information by analyzing control- and data-dependencies of the contract code. Then, it checks against both the predefined compliance and violation properties to detect vulnerabilities. SmartCheck [44] is an automated static code analyzer for smart contracts. It automatically checks smart contracts against a knowledge base for security vulnerabilities and bad practices. Slither [45] is a static analysis framework for Solidity, which contains a suite of vulnerability detectors and also provides an API for developing custom analyses.

Symbolic Execution. Oyente [9] is the first tool to apply symbolic execution in finding potential vulnerabilities in smart contracts. It formulates the vulnerabilities as intra-procedural properties, and uses symbolic execution to check against these properties. TEETHER [23] focuses their analysis on the critical paths of a contract program. Specifically, a path is critical if it includes an instruction whose arguments can be controlled by an attacker. Once a critical path is found, TEETHER computes the path conditions and infers the corresponding attack sequences for triggering the vulnerability. In addition, TEETHER also requires that the value transmitted in the final `CALL` instruction is greater than the sum of all values sent to the contract. This is similar to our approach but imprecise, because it does not model the whole transaction. MAIAN [10] is designed to find three types of problematic contracts: the prodigal, greedy and suicidal contracts. It formulates these three types of problems as inter-procedural properties, and performs bounded inter-procedural symbolic execution to search for property violations. EthRacer [31] investigates a family of event-ordering bugs in smart contracts. These bugs are intimately related to the dynamic ordering of contract events, i.e., function calls. The technical challenge in detecting event-ordering bugs in smart contract is the inherent combinatorial blowup in the path and state space analysis, even for simple contracts. The authors propose to use partial-order reduction techniques, using automatically extracted *happens-before* relations along with several dynamic symbolic execution optimizations.

Formal Verification. There are also attempts to formally verify smart contracts using either model checking or theorem-proving [11], [28], [43], [46], [47], [48], [49], [50], [51], [52]. Zeus [11] first translates Solidity source code into LLVM [53] intermediate language, and then performs the verification with the SeaHorn verification framework [54]. Hirai [46] defines a formal semantic model for EVM using the Lem language, and proves safety properties of contract programs compiled to Lem, with the interactive theorem prover Isabelle/HOL. KEVM [47] is a semantic encoding of EVM bytecode in the K-framework based on the rewriting logic. VerX [50] is an automated verifier for proving functional properties of smart contracts. VerX addresses an important problem, as all real-world contracts must satisfy custom functional specifications. VerX combines three techniques, enabling it to automatically verify temporal properties of infinite state smart contracts: (1) reduction of temporal property verification to reachability checking, (2) a new symbolic execution engine for EVM that is precise and efficient for a practical fragment of smart contracts, and (3) delayed predicate abstraction which uses symbolic execution during transactions and abstraction at transaction boundaries. VERISOL [55] studies the safety and security of smart contracts in the Azure Blockchain Workbench, an enterprise Blockchain-as-a-Service offering from Microsoft. It formalizes the *semantic conformance* of smart contracts against a state machine model with access-control policies, and develops a highly-automated formal verifier for Solidity that can produce proofs as well as discover counterexamples.

Static analysis approaches can be more efficient in terms of running time, but they often suffer from high false-positive rate. The main difference between our approach and these techniques is that, our approach dynamically executes the contract code on the real EVM environment, and therefore the detected vulnerabilities are guaranteed to be exploitable.

6.2 Dynamic Analysis

Some dynamic analysis techniques are proposed to address the vulnerabilities of smart contracts [56], [57].

Input generation. ContractFuzzer [8] is a fuzzing framework for detecting vulnerabilities of Ethereum smart contracts. It proposes seven specific patterns for seven types of vulnerabilities. Based on these patterns, it generates fuzzing inputs, instruments the EVM to collect the execution traces, and analyzes the traces to identify vulnerabilities. ReGuard [12] developed a fuzzing-based analyzer to automatically detect reentrancy vulnerabilities. Specially, it performs fuzz testing on smart contracts by iteratively generating random but diverse transactions. Based on the runtime traces, ReGuard dynamically identifies the reentrancy vulnerabilities. Echidna [58] takes a contract program as well as a set of invariants as input, and generates random inputs to trigger potential vulnerabilities. The invariants used by Echidna have to be written within the contract itself, thus they are not expressive enough to encode our inter-contract invariants. Wüstholz and Christakis [59] present a technique that extends greybox fuzzing with a method for learning new inputs based on already explored smart contract executions. The learned inputs can be used to guide exploration towards specific executions, for instance, ones that increase path coverage.

The main difference between our approach and other dynamic approaches is that we provide general principles that drill down to the very root of vulnerabilities, while other approaches use generic properties to detect specific vulnerabilities. By definition, fixed collections of properties are limited and modeled at the syntactic level; thus, they usually suffer from both false negatives and false positives. We believe that the absence of a general and precise test oracle is the main reason that there exist very few dynamic techniques for detecting vulnerabilities in smart contracts.

Test oracle. The lack of a precise test oracle is often the main bottleneck in software test automation [60]. An empirical analysis of model-derived test cases for Java programs shows that using a test oracle roughly doubles the defect detection rate [61]. Most activities to support the test oracle focus on providing better specification mechanisms, or on mining properties from the documentation or comments [60]. An *implicit test oracle* covers assumptions that have to hold globally for well-defined applications, e.g., no memory access to unallocated or uninitialized memory should ever happen. Sereum [62] protects the deployed smart contracts from being exploited. It addresses this problem in the context of re-entrancy exploits and propose a novel smart contract security technology, which protects existing, deployed contracts against re-entrancy attacks in a backwards compatible way based on run-time monitoring and validation. Sereum does not require any modification to or any semantic knowledge of existing contracts. Our work is also within that domain, as we cover the implicit assumption that no funds are created or destroyed by transactions. We therefore provide a valuable contribution in a field where it is in general very difficult to find useful implicit assumptions [60].

In general, specifications are provided by developers, either on a case-by-case based in code, or as more general rules that apply throughout the program. Specifications can be provided as executable code in the form of software design requirement, as preconditions, invariants, and post-conditions [63]. These facilities have been made available in the Solidity language as of version 0.4.10 [64], [65], but are not widely used yet. Compared to verification on traditional platforms, these features on Solidity have the drawback that their usage incurs side effects (in terms of the gas cost of computing the expression being evaluated); in general, side effects should be avoided in such expressions [66].

7 Conclusion

We propose ContraMaster, a grey-box fuzzing approach for finding exploitable vulnerabilities in smart contracts. Different from previous works, the proposed test oracle captures the very roots of transaction-related vulnerabilities based on invariants (Definitions 3 and 4), which are essential and not specific to any particular attack pattern. We also use feedback computed from the efficient runtime monitoring on EVM to guide the mutation of transaction sequences for fuzzing. We have demonstrated that ContraMaster is effective in finding exploitable vulnerabilities and produces much fewer false positives than the state-of-the-art. Furthermore, we find and confirm three new attacks.

Acknowledgments

This research is partly supported by the Ministry of Education, Singapore, under its Academic Rese arch Fund Tier 2 (MOE2018-T2-1-068), the National Research Foundation, Singapore, and the Energy Ma rket Authority, under its Energy Program (EP Award NRF2017EWT-EP003-023), and the Joint NTU-WeBank Research Centre of Eco-Intelligent Applications (THEIA), Nanyang Technological University, Singapor e (NWJ-2019-003).

References

[1] "Bitcoin Project," https://bitcoin.org/, 2019.
[2] "Ethereum Project," https://www.ethereum.org/, 2019.
[3] G. W. Peters and E. Panayi, "Understanding Modern Banking Ledgers Through Blockchain Technologies: Future of Transaction Processing and Smart Contracts on the Internet of Money," in *Banking Beyond Banks and Money*. Springer, 2016, pp. 239–278.
[4] R. Xu, L. Zhang, H. Zhao, and Y. Peng, "Design of Network Media's Digital Rights Management Scheme Based on Blockchain Technology," in *International Symposium on Autonomous Decentralized System*. IEEE, 2017, pp. 128–133.
[5] M. Iansiti and K. R. Lakhani, "The Truth about Blockchain," *Harvard Business Review*, 2017.
[6] J. Chang, B. Gao, H. Xiao, J. Sun, and Z. Yang, "sCompile: Critical Path Identification and Analysis for Smart Contracts," *arXiv preprint arXiv:1808.00624*, 2018.
[7] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical Security Analysis of Smart Contracts," in *ACM Conference on Computer and Communications Security*, 2018, pp. 67–82.
[8] B. Jiang, Y. Liu, and W. Chan, "ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 259–269.
[9] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in *ACM Conference on Computer and Communications Security*. ACM, 2016, pp. 254–269.
[10] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the Greedy, Prodigal, and Suicidal Contracts at Scale," *arXiv preprint arXiv:1802.06038*, 2018.
[11] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing Safety of Smart Contracts," in *The Network and Distributed System Security Symposium*, 2018.
[12] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "ReGuard: finding reentrancy bugs in smart contracts," in *ACM/IEEE International Conference on Software Engineering*. ACM, 2018, pp. 65–68.
[13] "DaoChallenge Source," https://etherscan.io/address/0x80f1f62b8b365c5326100d462d8570771b8d0e57, 2019, accessed 2019.
[14] Y. Feng, E. Torlak, and R. Bodík, "Precise Attack Synthesis for Smart Contracts," *arXiv preprint arXiv:1902.06067*, vol. abs/1902.06067, 2019.
[15] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/, 2016.
[16] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, p. 116, 2018.
[17] M. Jakobsson and A. Juels, "Proofs of Work and Bread Pudding Protocols," in *Secure Information Networks*. Springer, 1999, pp. 258–272.
[18] S. King and S. Nadal, "PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake," https://bitcoin.peryaudo.org/vendor/peercoin-paper.pdf.

[19] S. Wu, Y. Chen, Q. Wang, M. Li, C. Wang, and X. Luo, "CReam: A Smart Contract Enabled Collusion-Resistant E-Auction," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 7, pp. 1687–1701, 2018.
[20] "Solidity Document," https://solidity.readthedocs.io/en/v0.5.3/contracts.html#fallback-function.
[21] "Solidity," https://solidity.readthedocs.io/en/v0.5.1/, 2018.
[22] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun, "Executable Operational Semantics of Solidity," *arXiv preprint arXiv:1804.01295*, 2018.
[23] J. Krupp and C. Rossow, "teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts," in *27th USENIX Security Symposium USENIX Security*. ACM, 2018, pp. 1317–1333.
[24] K. Chatterjee, A. K. Goharshady, and Y. Velner, "Quantitative Analysis of Smart Contracts," in *European Symposium on Programming*. Springer, 2017, pp. 494–509.
[25] "ERC-20," https://theethereum.wiki/w/index.php/ERC20_Token_Standard/, 2018.
[26] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts," in *Principles of Security and Trust*. Berlin, Heidelberg: Springer, 2017, pp. 164–186.
[27] "Decentralized Application Security Project (or DASP) Top 10 of 2018," http://www.dasp.co/, 2018, accessed 2018.
[28] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking*. MIT press, 2018.
[29] T. Bernardi, N. Dor, A. Fedotov, S. Grossman, N. Immerman, D. Jackson, A. Nutz, L. Oppenheim, O. Pistiner, N. Rinetzky *et al.*, "Wip: Finding bugs automatically in smart contracts with parameterized invariants," 2020.
[30] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-State Based Binary Fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 627–637.
[31] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "Exploiting the Laws of Order in Smart Contracts," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2019, pp. 363–373.
[32] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *The Network and Distributed System Security Symposium*, vol. 17, 2017, pp. 1–14.
[33] G. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger. Ethereum Project Yellow Paper (2018)," 2018.
[34] "King of the Ether Throne," http://blockchain.unica.it/projects/ethereum-survey.
[35] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang, "Dependence Guided Symbolic Execution," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 252–271, 2017.
[36] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, Jul. 1987.
[37] Yoichi Hirai, "Integer Overflow/Underflow," https://github.com/ethereum/solidity/issues/796#issuecomment-253578925, 2016, accessed 2019.
[38] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2123–2138.
[39] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," *arXiv preprint arXiv:1812.11875*, 2018.
[40] NCC Group, "DASP: Decentralized Application Security Project," https://www.dasp.co/, 2019.
[41] Sjors Provoost, "Dao Challenge," https://medium.com/@dao.challenge/challenge-3-how-i-almost-lost-100-1a11a9824ccb, 2016, accessed 2019.
[42] Vitalik Buterin, "Ether Total Supply," https://github.com/ethereum/EIPs/issues/960, 2018, accessed 2019.
[43] K. Bhargavan, A. Delignat-Lavaud, and e. a. Fournet, "Formal Verification of Smart Contracts: Short Paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 2016, pp. 91–96.
[44] "smartCheck," https://tool.smartdec.net/, 2019.
[45] Trail of Bits, "Slither," https://github.com/trailofbits/slither, 2019, accessed 2019.
[46] Y. Hirai, "Defining the Ethereum Virtual Machine for Interactive Theorem Provers," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 520–535.
[47] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, "KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine," in *2018 IEEE 31st Computer Security Foundations Symposium*. IEEE, 2018, pp. 204–217.
[48] FStarLang, "Fstartlang," https://www.fstar-lang.org/, 2019, accessed 2019.
[49] Why3, "Why3," http://why3.lri.fr/, 2019, accessed 2019.
[50] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," 2020.
[51] D. Harz and W. Knottenbelt, "Towards safer smart contracts: A survey of languages and verification methods," *arXiv preprint arXiv:1809.09805*, 2018.
[52] J. Liu and Z. Liu, "A Survey on Security Verification of Blockchain Smart Contracts," *IEEE Access*, 2019.
[53] The LLVM Foundation, "LLVM," https://llvm.org/, accessed 2019.
[54] "SeaHorn," http://seahorn.github.io/, 2019.
[55] S. K. Lahiri, S. Chen, Y. Wang, and I. Dillig, "Formal Specification and Verification of Smart Contracts for Azure Blockchain," *arXiv preprint arXiv:1812.08829*, 2018.
[56] P. Zheng, Z. Zheng, X. Luo, X. Chen, and X. Liu, "A Detailed and Real-Time Performance Monitoring Framework for Blockchain Systems," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2018, pp. 134–143.
[57] Y. Liu, Y. Li, S.-W. Lin, and Q. Yan, "ModCon: A model-based testing platform for smart contracts," in *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Nov. 2020.
[58] Trail of Bits, "Echidna," https://github.com/trailofbits/echidna, 2019, accessed 2019.
[59] V. Wüstholz and M. Christakis, "Learning Inputs in Greybox Fuzzing," *arXiv preprint arXiv:1807.07875*, 2018.
[60] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2015.
[61] N. Li and J. Offutt, "An Empirical Analysis of Test Oracle Strategies for Model-Based Testing," in *IEEE Seventh International Conference on Software Testing, Verification and Validation*, March 2014, pp. 363–372.
[62] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," 2019.
[63] B. Meyer, "Applying 'Design by Contract'," *Computer*, vol. 25, pp. 40–51, 1992.
[64] "Smart Contract Security Best Practices," https://github.com/ConsenSys/smart-contract-best-practices, 2019, accessed 2019.
[65] Ethereum, "Security Considerations," https://solidity.readthedocs.io/en/develop/security-considerations.html, 2019, accessed 2019.
[66] P. Fritzson, M. Auguston, and N. Shahmehri, "Using Assertions in Declarative and Operational Models for Automated Debugging," *Journal of Systems and Software*, vol. 25, no. 3, pp. 223–239, 1994.

Haijun Wang received the Ph.D. degree in the School of Electronic and Information from Xi'an Jiaotong University, China, in 2016. He is currently working as post-doctoral research in school of computer science and engineering of Nanyang Technological University. His research interests include program analysis, regression testing, reverse engineering, software security, blockchain security.

Ye Liu is a Ph.D. student at the School of Computer Science and Engineering, Nanyang Technological University (NTU). He received his M.Sc. from the Beihang University in Jan 2019, and B.E. from the Northeastern University (China) in June 2016. Later, he worked for half a year as a research assistant in the cybersecurity lab at NTU. He started his Ph.D. study from Aug 2019, supervised by Prof. Yi Li and Prof. Yang Liu. His research interests mainly focus on smart contract reliability and security.

Yi Li is an Assistant Professor from the School of Computer Science and Engineering at the Nanyang Technological University, Singapore. He received his M.Sc. and Ph.D. in Computer Science from the University of Toronto in 2013 and 2018. Yi's research interests are in software engineering, software security, and software sustainability. His recent work on software history analysis won an ACM Distinguished Paper Award at the 30th International Conference on Automated Software Engineering (ASE'15). He was the PC Co-Chair of ICECCS'20 and the ICFEM'19 Doctoral Symposium, and served as a member of the program committee for FSE'21, FASE'21, ASE'20, and ICDCS'20.

Shang-Wei Lin is an Assistant Professor in School of Computer Science and Engineering of Nanyang Technological University (NTU). He received his Ph.D. degree in Computer Science and Information Engineering from the National Chung Cheng University in 2010. In 2011, he was a postdoctoral researcher in School of Computing, National University of Singapore (NUS). From 2012 to November 2014, he was a research scientist at Temasek Laboratories in National University of Singapore (NUS). From December 2014 to April 2015, he was a postdoctoral research fellow in Singapore University of Technology and Design (SUTD). He has joined NTU as Assistant Professor in April 2015. His research interests include formal verification, formal synthesis, embedded system design, cyber-physical systems, security systems, multi-core programming, and component-based object-oriented application frameworks for real-time embedded systems. Recently, he is working on applying formal methods on smart contract verification.

Cyrille Artho is an Associate Professor at the KTH Royal Institute of Technology, Stockholm. His main interests are software verification and software engineering, in particular for concurrent and networked systems. In his Master's thesis and his Ph.D. thesis at ETH Zurich, he investigated different approaches for finding faults in multi-threaded programs. During that time, in joint work with NASA Ames, he found that high-level data races can show potential problems in concurrent software even if individual data accesses are safe. After obtaining his Ph.D., he moved to Tokyo, where he worked for two years at the National Institute of Informatics as a Postdoctoral Researcher before working at the National Institute of Advanced Industrial Science and Technology (AIST) in Tokyo and Osaka until July 2016.

Lei Ma is a tenured Assistant Professor with Information Science and Electrical Engineering, Kyushu University, Japan. He received Ph.D. and M.E. from The University of Tokyo in 2014 and 2011, and B.E. from Shanghai JiaoTong University in 2009. His current research focues on interdisciplinary research fields of Software Engineering, Security and Artificial Intelligence, and especially on proposing quality assurance and security solutions for machine learning engineering. Over the past several years, he has made more than 30 top tier conference/journal papers relevant to software testing. In particular, he won the championship of 2015 IEEE International Search Based Software Testing Contest, and received 7 other academic awards, including 2 ACM SIGSOFT Distinguished Paper Award (ASE'15, ASE'18), a FOSS Impact Paper Award (MSR'18), a Best Candidate Paper Award (SANER'16), a Best Paper Award (IEEE HotWeb'15), etc.

Yang Liu graduated in 2005 with a Bachelor of Computing (Honours) in the National University of Singapore (NUS). In 2010, he obtained his Ph.D. and started his post-doctoral work in NUS and MIT. In 2012, he joined Nanyang Technological University (NTU), and currently is a full professor and Director of the cybersecurity lab in NTU. Dr. Liu specializes in software verification, security and software engineering. His research has bridged the gap between the theory and practical usage of formal methods and program analysis to evaluate the design and implementation of software for high assurance and security. By now, he has more than 270 publications in top tier conferences and journals. He has received a number of prestigious awards including MSRA Fellowship, TRF Fellowship, Nanyang Assistant Professor, Tan Chin Tuan Fellowship, Nanyang Research Award and 8 best paper awards in top conferences like ASE, FSE and ICSE.