DEFECTCHECKER: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode

Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo and Ting Chen

Abstract—Smart contracts are Turing-complete programs running on the blockchain. They are immutable and cannot be modified, even when bugs are detected. Therefore, ensuring smart contracts are bug-free and well-designed before deploying them to the blockchain is extremely important. A contract defect is an error, flaw or fault in a smart contract that causes it to produce an incorrect or unexpected result, or to behave in unintended ways. Detecting and removing contract defects can avoid potential bugs and make programs more robust. Our previous work defined 20 contract defects for smart contracts and divided them into five impact levels. According to our classification, contract defects with seriousness level between 1-3 can lead to unwanted behaviors, e.g., a contract being controlled by attackers. In this paper, we propose *DefectChecker*, a symbolic execution-based approach and tool to detect eight contract defects that can cause unwanted behaviors of smart contracts on the Ethereum blockchain platform. *DefectChecker* can detect contract defects from smart contracts' bytecode. We compare *DefectChecker* with key previous works, including *Oyente, Mythril* and *Securify* by using an open-source dataset. Our experimental results show that *DefectChecker* performs much better than these tools in terms of both speed and accuracy. We also applied *DefectChecker* to 165,621 distinct smart contracts on the Ethereum platform. We found that 25,815 of these smart contracts contain at least one of the contract defects that belongs to impact level 1-3, including some real-world attacks.

Index Terms—Smart Contracts, Ethereum, Contract Defects Detection, Bytecode Analyze, Symbolic Execution

1 INTRODUCTION

In recent years, decentralized cryptocurrencies have attracted considerable interest. To ensure these systems are scalable and secure without the governance of a centralized organization, decentralized cryptocurrencies adopt the *blockchain* concept as their underlying technology. Bitcoin [1] was the first digital currency, and it allows users to encode scripts for processing transactions automatically. However, scripts in Bitcoin are not Turing-complete, which restricts their application to currencies, such as money transfer or payment. To address this limitation, Ethereum [2] leverages a technology named *Smart Contracts*, which are Turing-complete programs that run on the blockchain. By utilizing this technology, practitioners can develop decentralized applications (DApps) [3] and apply blockchain techniques to different fields such as gaming [4] and finance [5].

Smart contracts are usually developed using a high-level programming language, such as Solidity [6]. When developers deploy a smart contract to Ethereum, the contract will first be compiled into Ethereum Virtual Machine (EVM) *bytecode*. Then, each node on the Ethereum system will receive the smart contract bytecode and have a copy in their ledger. Anyone, even attackers, can invoke the smart contract by sending transactions to the corresponding contract address.

Key features of smart contracts make them become attractive targets for hackers [7]. On the one hand, many smart contracts hold valuable Ethers, and they cannot hide their balance, which gives financial motivation for attacks by hackers [8], [9]. On the other hand, smart contracts run in a permission-less network, which means hackers can check all the transactions and bytecode freely, and try to find bugs on the contracts. Even worse, smart contracts cannot be modified, even when bugs are detected. Therefore, ensuring smart contracts are bug-free and well-designed before deploying them to Ethereum is extremely important.

A contract defect [10], [11] is an error, flaw, or fault in a smart contract that causes it to produce an incorrect or unexpected result, or to behave in unintended ways [12]. The detection and removal of contract defects is a method to avoid potential bugs and improve the design of existing code. In our previous work [11], we first defined 20 contract defects by analyzing StackExchange [13] posts. It is also the first work that used an online survey to validate whether smart contract developers consider these contract defects as harmful, which make the definitions more persuasive. The work divided the defined 20 contract defects into five impact levels and showed that smart contracts contain defects with impact levels 1 to 3 can lead to unwanted behaviors, e.g., contracts being controlled by attackers.

However, our previous work did not propose a suitable tool that could detect these contract defects. To address this limitation, in this paper, we propose *DefectChecker* to detect

- *Jiachi Chen, Xin Xia and John Grundy are with the Faculty of Information Technology, Monash University, Melbourne, Australia. E-mail: {Jiachi.Chen, Xin.Xia, John.Grundy}@monash.edu*
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore. E-mail: davidlo@smu.edu.sg*
- *Xiapu Luo is with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong. E-mail: csxluo@comp.polyu.edu.hk*
- *Ting Chen is with the School of Computer Science and Engineering, University of Electronic Science and Technology of China, China. E-mail: brokendragon@uestc.edu.cn*
- *Xin Xia is the corresponding author.*

Manuscript received ; revised

eight contract defects defined in our previous work that belong to serious impact level 1 (high) to level 3 (medium), by using the bytecode of smart contracts. *DefectChecker* symbolically executes the smart contract through bytecode, and without the needs of source code. During the symbolic execution, *DefectChecker* generates the CFG of smart contracts, as well as the "stack event", and identifies three features, i.e., "Money Call", "Loop Block", and "Payable Function". By using the CFG, stack event, and the three features, we design eight rules to detect each contract defect.

We verify the performance of *DefectChecker* by applying it to an open-source dataset developed in our previous work [11]. We also compare its results with those of three state-of-the-art tools, i.e., *Oyente, Mythril* and *Securify*. Our evaluation results show that *DefectChecker* obtains the highest F-score (88.8% in the whole dataset) and requires the least time (0.15s per contract) to analyze one smart contract compared to these other baseline tools. We also crawled all of the bytecode of smart contracts deployed on Ethereum by Jan. 2019 and applied *DefectChecker* to these *165,621* distinct bytecode smart contracts. We found that *15.9%* of smart contracts on Ethereum contain at least one of contract defects (the severity level 1 to 3) using *DefectChecker*.

The main contributions of this work are:

- To the best of our knowledge, *DefectChecker* is the most **accurate** and the **fastest** symbolic execution-based model for smart contract defects detection.
- We systematically evaluated our tool using an open source dataset to test its performance. In addition, we crawled all of the bytecode (165,621) on the Ethereum platform by the time of writing the paper and identified 25,815 smart contracts that contain at least one contract defect. Using these results, we find some real-world attacks, and give examples to show the importance of detecting contract defects.
- Our datasets, tool and analysis results have been released to the community at https://github.com/Jiachi-Chen/DefectChecker/.

The organization of the rest of this paper is as follows. In Section 2, we provide background knowledge of smart contracts and introduce eight contract defects with code examples. Then, we introduce the architecture of DEFECTCHECKER in section 3 and present its evaluation in section 4. We conduct a large scale evaluation based on Ethereum smart contracts in Section 5 and give two real-world attacks as case studies. In section 6, we introduce the related works. Finally, we conclude the study and discuss possible future work in Section 7.

2 BACKGROUND AND MOTIVATION

In this section, we briefly introduce key background information about smart contracts and their contract defects.

2.1 Smart Contracts

Contracts. Leveraging blockchain techniques, smart contracts are autonomous protocols stored on the blockchain. Once started, the running of a contract is automatic and it runs according to the program logic defined beforehand [14]. When developers deploy a smart contract to Ethereum, the contract will be compiled to EVM bytecode and identified by a unique 160-bit hexadecimal hash contract address. The smart contract execution depends on their code, and even the creator cannot affect its running or state. For example, if a contract does not contain functions for Ether transfer, even the creator cannot withdraw the Ethers. Smart contracts run on a permission-less network. Anyone can invoke the methods of smart contracts through ABI (Application Binary Interface) [6]. The contract bytecode, transactions, and invocation parameters are visible to everyone.

Gas System. To ensure the security of smart contracts, each transaction of a smart contract will be run by all miners. Ethereum uses the *gas system* [15] to measure its computational effort, and the developers who send transactions to invoke smart contracts need to pay an execution fee. The execution fee is computed by: $gas\_cost \times gas\_price$. Gas cost depends on the computational resource that takes by the execution and gas price is offered by the transaction creators. To limit gas cost, when developers send their transactions to invoke contracts, they will set the *Gas Limit* which determines the maximum gas cost. If the gas cost of a transaction exceeds its *Gas Limit*, the execution will fail and throw an *out-of-gas error* [2]. There are some special operations which will limit the *Gas Limit* to a specific value. For example, *address.transfer()* and *address.send()* are two methods provided by Ethereum that are used to send Ethers. If a smart contract uses these methods to send Ethers to another smart contract, the *Gas Limit* will be restricted to 2300 gas units [6]. 2300 gas units are not enough to write to storage, call functions or send Ethers, which can lead to the failure of transactions. Therefore, *address.transfer()* and *address.send()* can only be used to send Ethers to *external owned accounts* (EOA). (There are two types of accounts on Ethereum: externally owned accounts which controlled by private keys, and contract accounts which controlled by their contract code [2].)

Ethereum Virtual Machine (EVM). To deploy a smart contract to Ethereum, its source code needs to be compiled to bytecode and stored on the blockchain. EVM is a stack-based machine; when a transaction needs to be executed, EVM will first split bytecode into bytes; each byte represents a unique instruction called opcode. There are 140 unique opcodes by April 2019 [2], and each opcode is represented by a hexadecimal number [2]. EVM uses these opcodes to execute the task. For example, consider a bytecode 0x6070604001. EVM first splits this bytecode into bytes (0x60, 0x70, 0x60, 0x40, 0x01), and execute the first byte 0x60, which refers to opcode *PUSH1*. *PUSH1* pushes one byte data to EVM stack. Therefore, 0x70 is pushed to the stack. Then, EVM reads the next 0x60 and push 0x40 into the stack. Finally, EVM executes 0x01, which refers to opcode *ADD*. *ADD* obtains the next two values from the top of the stack, i.e., 0x70 and 0x40, and put their sum (B0), a hex result into the stack.

EVM Bytecode v.s. JVM Bytecode in Control Flow Analysis. Control flow analysis methods have been widely used in other stack-based machines, e.g., JVM [16]. However, there are many differences in analyzing the control flow of Java bytecode and EVM bytecode. These differences increase some new challenges in analyzing EVM bytecode. We highlight the difference between EVM bytecode analyze method

we used in this paper and JVM bytecode, which includes: (1) JVM bytecode has a fixed stack depth under different control-flow paths. The execution of JVM cannot reach the same program point with different stack sizes. [17] There are no such constraints for EVM bytecode, which increase the difficulty of identifying the control-flow constructs of EVM bytecode. (2) The jump target of EVM bytecode is read from EVM stack. When conditional jump is used, the target will be affected by the second stack item. Therefore, we need to symbolic execute the EVM bytecode to construct the control-flow edges. In contrast, Java bytecode has a clearly defined set of targets of every jump [18] (3) JVM bytecode has defined method invocation and return instructions. In contrast, EVM bytecode uses jumps to realize the intra-contract function calls. In this case, to resolve an intra-contract function call, we need to inspect the top stack element to determine the jump target. Moreover, we can identify the inter-contract function calls by inspecting specific opcodes, such as CALL, CALLCODE, DELEGATECALL, STATICCALL, etc. [6], [18]

The Fallback Function. The fallback function is a unique feature of smart contracts compared to traditional programs. An example can be found at Line 13 of Listing 4, which is the only unnamed function in smart contracts programming [6]. The fallback function does not have any arguments or return values. It will be executed automatically on a call to the contract if none of the functions match the given function identifier [6]. For example, if a transaction calls function 'A' of the contract, while there is no function named 'A'. In this situation, fallback function will automatically be executed to handle the invocation. If the function is marked by *payable* [6], the fallback function will also be executed automatically when receiving Ethers.

The Call Instruction and Ether Transfer. Ether transfer is an important feature on Ethereum. In Solidity programming, there are three methods to transfer Ethers, i.e., *address.call.value()*, *address.transfer()*, and *address.send()*. Among these three methods, only *address.call.value()* allows users to send Ethers to a contract address, as the other two methods are limited to 2300 gas units, which are not enough to send Ethers. *address.send()* returns a boolean value, while *address.transfer()* throws an exception when errors happen and returns nothing. All of these three methods can generate a CALL instruction in contract bytecode. Other behaviors, e.g., function call, can also generate CALL instructions. A CALL instruction reads seven values from the top of EVM stack. They represent the gas limitation, recipient address, transfer amount, input data start position, size of the input data, output data start position, size of the output data, respectively.

2.2 Contract Defects in Smart Contracts

Our previous work [11] defined 20 contract defects for smart contracts. We divided these contract defects into five "impact" levels; among these contract defects, 11 belong to impact level one (most serious) to three (low seriousness) that might lead to unwanted behaviors. The definition of these 11 contract defects is given in Table 1. In this paper, we propose *DefectChecker*, a symbolic execution tool to detect eight of these impact level one to three contract defects. *DefectChecker* does not detect contract defects belonging to levels 4 and 5, as these contract defects will not affect the normal running of the smart contracts according to the definition. For example, *Unspecified Compiler Version* is one of the level 5 smart contract defects. The removal of the contract defects requires the developer of the contract to use a specific compiler like 0.4.25. This contract defect will not affect the normal running of the contract and will only pose a threat for code reuse in the future. This kind of contract defect is also difficult to detect at the bytecode level as much semantic information is lost after compilation.

However, please note that in this work, we do not consider three of the contract defects that belong to impact level 1 to 3 – *Unmatched Type Assignment*, *Hard Code Address* and *Misleading Data Location*, as they are not easy to detect at bytecode level. Our analysis shows that they appear 22, 84, and 1 times among 587 smart contacts, respectively. EVM will remove or add some information when compiling smart contracts to bytecode, which may cover up these taints on the source contract code. For ***Hard Code Address***, the bytecode we obtain from the blockchain does not contain information on the *construct* function, while we found most *Hard Code Address* errors appear in *construct* functions. To detect ***Unmatched Type Assignment***, we need to know the maximum loop iterations, which is usually read from storage, and is not easy to obtain the value through static analysis. For example, for a loop "*for(uint8 i = 0; i < num; i++)*", the data range of uint8 is from 0 to 255. Thus, if num is larger than 255, the loop will overflow. However, num is usually a storage variable which is read from storage or depends on an external input. Thus, it is difficult to detect this through bytecode analysis. ***Misleading Data Location*** is also not easy to detect from bytecode. In Solidity programming, *storage* in Solidity is not dynamically allocated and the type of *struct*, *array* or *mapping* are maintained on the storage. Thus, these three types created inside a function can point to the *storage slot* 0 by default, which can lead to potential bugs. However, we cannot know whether the point on slot 0 is correct or a mistake made by EVM.

2.2.1 Definition of Impact Levels

Below we give representative concrete examples of each of the eight smart contract defects, and introduce the definition of impact level one to three according to our previous work.

- **Impact 1 (IP1):** Smart contracts containing these contract defects can lead to critical unwanted behaviors. Unwanted behaviors can be triggered by attackers, and they can make profits by utilizing the defects.
- **Impact 2 (IP2):** Smart contracts containing these contract defects can lead to critical unwanted behaviors. Unwanted behaviors can be triggered by attackers, but they cannot make profits by utilizing the defects.
- **Impact 3 (IP3):** There are two types of IP3. **Type A**: Smart contracts containing these contract defects can lead to critical unwanted behaviors, but unwanted behaviors cannot be triggered by attackers. **Type B**: Smart contracts containing these contract defects can lead to major unwanted behaviors. The unwanted behaviors can be triggered by attackers, but they cannot make profits by utilizing the defects.

TABLE 1: The Definitions of contract defects with Impact level 1-3. The first eight contract defects can be detected by *DefectChecker*.

Contract Defect	Definition	Impact Level	Contract Defect	Definition	Impact Level
Transaction State Dependency (TSD)	Using tx.origin to check the permission.	IP1	*DoS Under External Influence (DuEI)*	Throwing exceptions inside a loop which can be influenced by external users	IP2
Strict Balance Equality (SBE)	Using strict balance quality to determine the execute logic.	IP2	*Reentrancy (RE)*	The re-entrancy bugs.	IP1
Nested Call (NC)	Executing CALL instruction inside an unlimited-length loop.	IP2	*Greedy Contract (GC)*	A contract can receive Ethers but can not withdraw Ethers.	IP3
Unchecked External Calls (UEC)	Do not check the return value of external call functions.	IP3	*Block Info Dependency (BID)*	Using block information related functions to determine the execute logic.	IP3
Unmatched Type Assignment	Assigning unmatched type to a value, which can lead to integer overflow	IP2	*Misleading Data Location*	The reference types of local variables with *struct*, *array* or *mapping* do not clarify	IP2
Hard Code Address	Using hard code address inside smart contracts.	IP3			

Critical represents contract defects, which can lead to the crash, being controlled by attackers, or can lose all the Ethers. Major represents the contract defects that can lead to the loss of a part of the Ethers [11].

2.2.2 Examples of Smart Contract Defects

```
1  contract Victim { ...
2    address owner = owner_address;
3    function sendMoney(address addr){
4      require(tx.origin == owner);
5      addr.transfer(1 Ether);
6    }
7  }
8  contract Attacker{ ...
9    function attack(address vim_addr,address myAddr){
10     Victim vic = Victim(vim_addr);
11     vic.sendMoney(myAddr);
12   }
13 }
```

Listing 1: Transaction State Dependency

(1). Transaction State Dependency (TSD): Contracts need to check whether the caller has the right permission for some permission sensitive functions. The failure of the permission check can cause serious consequences. *tx.origin* can get the original address of the transaction, but this method is not reliable as the address returned by this method depends on the transaction state. Therefore, *tx.origin* should not be used to check whether the caller has permission to execute functions.

Example: In Listing 1, The *Attacker* contract can make a permission check fail by utilizing the *attack* function (Line 9). By utilizing this method, anyone can execute *sendMoney* function (Line 3) and withdraw the Ethers in the contract.

Possible Solution: *Solidity* provides *msg.sender* to obtain the sender address, which can be used to check permissions instead of using *tx.origin*.

(2). DoS under External Influence (DuEI): Smart contracts will rollback a transaction if exceptions are detected during their running. If the error that leads to the exception cannot be fixed, the function will give a denial of service (*DoS*) error perpetually.

Example: Listing 2 shows such an example. Here, *members* is an array which stores many addresses. However, one of the address is an attacker contract, and the transfer function can trigger an out-of-gas exception due to the 2300 gas limitation [2]. Then, the contract state will rollback. Since the code cannot be modified, the contract can not remove the attack address from *members* list, which means that if the attacker does not stop attacking, the following function cannot work anymore.

Possible Solution: Developers can use a boolean value check instead of throwing exceptions in the loop. For example, using "*if(members[i].send(0.1 ether) == false) break;*" instead of line 3 in listing 2.

```
1 for(uint i = 0;i < members.length; i++){
2     if(this.balance > 0.1 ether)
3         members[i].transfer(0.1 ether);
4 }
```

Listing 2: DoS under External Influence

(3). Strict Balance Equality (SBE): Attackers can send Ethers to any contracts forcibly by utilizing *selfdestruct()* [6]. This method will not trigger the fallback function, which means the victim contract cannot reject the Ethers. Therefore, smart contract logic may fail to work due to the unexpected Ethers sent by attackers.

Example: The *doingSomething()* function in listing 3 can only be triggered when the balance strict equal to 1 *ETH*. However, the attacker can send 1 *Wei* (1 ETH = 1e18 Wei) to the contract to make the balance never equal to 1 ETH.

Possible Solution: The contract can use "$\geq$" to replace "==" as attackers can only add to the amount of a balance. In this case, it is difficult for the attackers to affect the logic of the program.

```
1 if(this.balance == 1 eth)  doingSomething();
```

Listing 3: Strict Balance Equality:

(4). Reentrancy (RE): In Ethereum, a function can be executed several times in one execution by using the *Call* method. When a contract calls another, the execution waits for the call to finish [19]. Thus, it can lead to multiple invocations and money transfer in some situations.

Example: Listing 4 shows an example of a reentrancy defect. There are two smart contracts, i.e., *Victim* contract and *Attacker* contract. The *Attacker* contract is used to transfer Ethers from *Victim* contract, and the *Victim* contract can be regarded as a bank, which stores the Ethers of users. Users can withdraw their Ethers by invoking *withdraw()* function, which contains Reentrancy defects.

First, the *Attacker* contract uses the *reentrancy()* function (L16) to invoke *Victim* contracts *withdraw()* function in line

3. The *addr* in line 16 is the address of the *Victim* contract. Normally, the *Victim* contract sends Ethers to the callee in line 6, and resets callees balance to 0 in line 7. However, the *Victim* contract sends Ethers to the *Attacker* contract before resetting the balance to 0. When the *Victim* contract sends Ethers to the *Attacker* contract (L6), the fallback function (L13) of the *Attacker* contract will be invoked automatically, and then invoking the *withdraw()* function (L14) again. The invoking sequence in this example is: L16 $\rightarrow$ L3 $\rightarrow$ L6 $\rightarrow$ L13 $\rightarrow$ L3 $\rightarrow$ L6 $\rightarrow$ L13 $\cdots$

```
1  contract Victim {    ...
2    mapping(address => uint) public userBalance;
3    function withdraw(){
4      uint amount = userBalance[msg.sender];
5      if(amount > 0){
6        msg.sender.call.value(amount)();
7        userBalance[msg.sender] = 0;
8        }
9    }
10   ...
11 }
12 contract Attacker{   ...
13   function() payable{
14     Victim(msg.sender).withdraw();
15   }
16   function reentrancy(address addr){
17     Victim(addr).withdraw();
18   }
19   ...
20 }
```

Listing 4: Reentrancy

Possible Solution: There are 3 kinds of *Call* methods that can be used to send Ethers in Ethereum, i.e., *address.send()*, *address.transfer()*, and *address.call.value()*. *address.send()* and *address.transfer()* will change the maximum gas limitation to 2300 gas units if the recipient is a contract account. 2300 gas units are not enough to transfer Ethers, which means *address.send()* and *address.transfer()* cannot lead to *Reentrancy*. Therefore, using *address.send()* and *address.transfer()* instead *address.call.value()* can avoid *Reentrancy*.

(5). Nested Call (NC): Instruction *CALL* is very expensive (9000 gas paid for a non-zero value transfer as part of the CALL operation) [2]. If a loop contains the *CALL* instruction but does not limit the loop iterations, the total gas cost may have a high risk to exceed its gas limitation.

Example: In listing 5, if we do not limit the loop iterations, attackers can maliciously increase its size to cause an out-of-gas error. Once the out-of-gas error happens, this function cannot work anymore, as there is no way to reduce the loop iterations.

Possible Solution: Developers should estimate the maximum loop iterations and limit the loop iterations.

```
1 for(uint i = 0; i < member.length; i++){
2     member[i].send(1 wei);
3 }
```

Listing 5: Nested Call

(6). Greedy Contract (GC): Ethers on smart contracts can only be withdrawn by sending Ethers to other accounts or using *selfdestruct* function. Otherwise, even the creators of the smart contracts cannot withdraw the Ethers and Ethers will be locked forever. We define that a contract is a greedy contract if the contract can receive Ethers (contains payable functions) but there is no way to withdraw the Ethers.

Example: Listing 6 is a greedy contract. The contract is able to receive Ethers as it contains a payable fallback function in line 2. However, the contract does not contain any methods to transfer money to others. Therefore, the Ethers on the contract will be locked forever.

Possible Solution: Adding a function to withdraw Ethers if the contract can receive Ethers.

```
1 Contract Greedy{
2     function() payable{
3         process(msg.sender);
4     }
5     function process(address addr) {...}
6 }
```

Listing 6: Greedy Contract

(7). Unchecked External Call (UEC): Solidity provides many functions (*address.send()*, *address.call()*) to transfer Ethers or call functions between contracts. However, these call-related methods can fail, e.g., have a network error or run out of gas. When errors happen, these functions will return a boolean value but never throw an exception. If the callers do not check the return values of the external calls, they cannot ensure whether the logic of the following code snippets is correct.

Example: Listing 7 shows such an example. Line 1 does not check the return value of the *address.send()*. As the Ether transfer can sometimes fail, line 1 cannot ensure whether the logic of the following code is correct.

Possible Solution: Always checking the return value of the *address.send()* and *address.call()*.

```
1 address.send(ethers); doingSomething();   //bad
2 if(address.send(ethers)) doingSomething(); //good
```

Listing 7: Unchecked External Call

(8). Block Info Dependency (BID): Developers can utilize a series of block related functions to obtain block information. For example, block.blockhash is used to obtain the hash number of the current block. Many smart contracts rely on these functions to decide a program's execution, e.g., generating random numbers. However, miners can influence block information, e.g, miners can vary the block time stamp by roughly 900 seconds [19]. In this case, the block info dependency operation can be controlled by miners to some extent.

Example: The contract in listing 8 is a code snippet of a roulette contract. The contract utilizes block hash number to select a winner, and send winner one Ether as bonus. However, the miner can control the result. So, the miner can always be the winner.

Possible Solution: The precondition of a safe random number is that the random number cannot be controlled by a single person, e.g., a miner. The completely random information we can use in Ethereum includes users' addresses, users' input numbers and so on. Also, it is important to hide the values used by the contract for other players to avoid attacks. Since we cannot hide the address of users and their submitted values on Ethereum, a possible solution to generate random numbers without using block related functions is using a hash number. The algorithm has three rounds:

Round 1: Users obtain a random number and generate a hash value in their local machine. The hash value can be obtained by *keccak256* function, which is a function provided by Ethereum. After obtaining the random number, users submit the hash number.

Round 2: After all the users submit the hash number, users are required to submit the original random number. The contract checks whether the original number can generate the same hash number by using the same *keccak256* function.

Round 3: If all users submit correct original numbers, the contract can use the original numbers to generate a random number.

```
1 address[] participators;
2 uint winnerID = uint(block.blockhash) %
      participators.length
3 participators[winnerID].transfer(1 eths);
```

Listing 8: Block Info Dependency:

3 THE *DefectChecker* APPROACH

3.1 Design Overview

Figure 1 depicts an overview architecture of the *DefectChecker* approach. There are four components of *DefectChecker*, i.e., *Inputter*, *CFG Builder*, *Feature Detector*, and *Defect Identifier*.

The left part of the figure is the Inputter, and users can feed bytecode as *input*. Solidity source code is also allowed, but it needs to be compiled into bytecode. Bytecode is then disassembled into opcodes by utilizing API provides by *Geth*. Then, *DefectChecker* splits opcode into several basic blocks and symbolically executes instructions in each block. After that, *DefectChecker* generates the CFG (control flow graph) of a smart contract and records all stack events. During symbolic execution, *Feature Detector* detects three features (i.e., Money Call, Loop Block and Payable Function), all concepts introduced below. Based on this information, *Defect Identifier* uses eight different rules to identify the contract defects on smart contracts.

Detecting contract defects by bytecode is very important for smart contracts on Ethereum. All the bytecode of smart contracts are stored on the blockchain, but only less than 1% of smart contracts have opened their source code [20]. Smart contracts usually call other contracts, but the callee contracts may not open their source code for inspection. In such a case, the caller smart contracts can only detect whether the callee contract is secure through their bytecode.

3.2 Basic Block Builder

A basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit [21]. We first split the opcode into several blocks and give a type of the block according to its exit type. The exit type can be determined by the last instruction on a block. If the last instruction is *JUMP* or *JUMPI*, the block type is *unconditional* or *conditional*, respectively. If the last instruction is a terminal instruction (*STOP*, *REVERT* and *RETURN*), the block type is *terminal*. Some blocks are none of these three types, we call their block type as *fall*. In summary, four types of blocks are classified, i.e., *unconditional*, *conditional*, *fall*, and *terminal*.

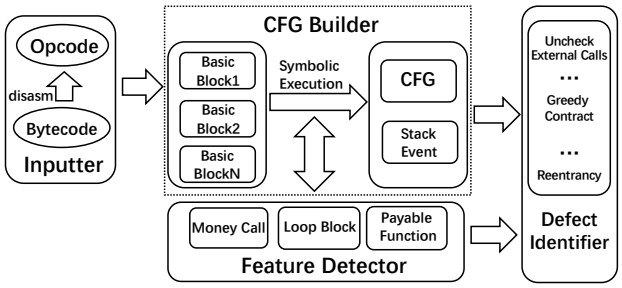

Fig. 1: Overview architecture of *DefectChecker*

3.3 Symbolic Execution

Unlike other stack-based machines, e.g., JVM where Java bytecode has a clearly-defined set of targets for every jump, the jump position of EVM bytecode needs to be calculated during symbolic execution. Thus, *DefectChecker* needs to symbolically execute each single EVM instruction one at a time to obtain the CFG for smart contracts. EVM is a stack-based machine – when executing an instruction, it reads several symbolic states from the top of the EVM stack and put the symbolic result back to the EVM stack. During the symbolic execution, we can obtain the jump relations between blocks. There are three types of block according to the jump behaviors, i.e., *conditional jump*, *unconditional jump* and *fall execution*. *Stack Event* records all symbolic states on the EVM stack after the execution of each instruction.

```
1 function example(uint num) returns(uint){
2     if(num > 10)
3         return 1;
4     else{
5         return 0;
6     }
7 }
```

Listing 9: Code of Figure 2

Figure 2 is an example of the symbolic execution of the code in Listing 9. There are 4 blocks in this figure, and each block contains several instructions. The instructions in block 1 represent the code *if(num >10)*. The block 2 and block 3 put the value (0 or 1) to the EVM stack, respectively. The instructions in *block 4* are used to return the value(0 or 1) to the environment. The left-most number in each line indicates instructions' index ID, and the center part is the instruction that needs to be executed. All the instructions will execute sequentially according to their index ID. If the instruction is 'PUSH', the right-most part will have a value that pushes into EVM stack. There is a Program Counter (PC) that records the ID that being executed at the current time. The PC starts from ID 0 in block 1, and EVM executes this instruction.

The example shown in Figure 2 is a part of the code of a contract, so the PC starts from index ID 130 in block 1. Before EVM executes the instruction *JUMPDEST*, there is a symbol *num* in the EVM stack. The symbol *num* represents the input value of the function (L1 of Listing 9). *JUMPDEST* marks a valid destination for jumps; it does not read or push any values. So the PC points to ID 131, and EVM pushes a value 0 to EVM stack. Then, '10' is pushed into EVM stack and PC point to 135. *DUP3* duplicates the 3rd stack

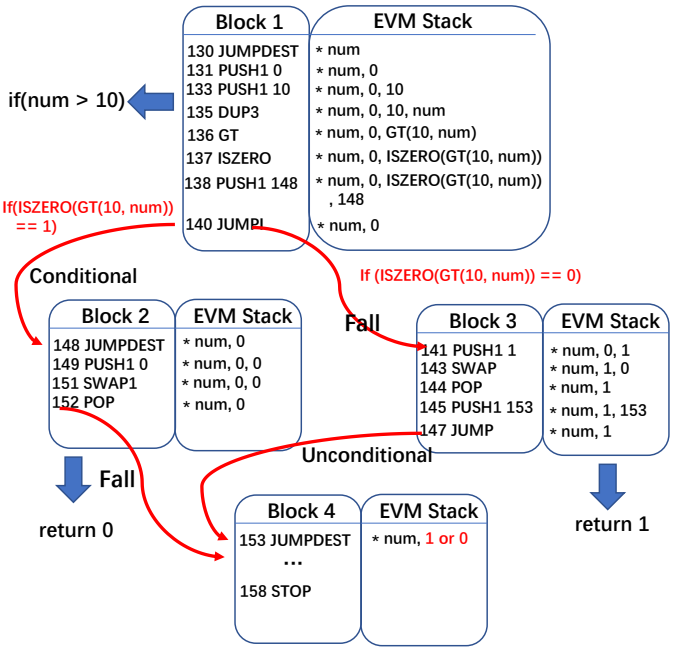

Fig. 2: Example of Symbolic Execution

item. Therefore, the symbol *num* is pushed into EVM stack. *GT* reads two values from the EVM stack. If the first value (at the top of the stack) is greater than the second value, than EVM push 1 into the stack; otherwise, 0 is pushed. We use a symbol *GT(a, num)* to represent the result and push the result into the EVM stack. Then, *ISZERO* reads a value from the top of the EVM stack. ISZERO reads one value from EVM. If the value equal to zero, then we push 1 into stack; otherwise, we push 0. We use a symbol *ISZERO(GT(a, num))* to represent the result and push the result into the EVM stack. *JUMPI* (ID 140) reads two values from the stack, the first value represents the jump position '148', and the second value is a conditional expression. If the result of the conditional expression is "1" (true), the the PC jumps to the index ID 148, which indicates the start position of block 2. Otherwise, if the result is "0" (false), the EVM falls to execute the following index ID 141(the start position of Block 3).

Since the result of *ISZERO(GT(a, num))* can be "0" or "1", this symbolic execution can generate two paths, i.e., Block 1 $\rightarrow$ Block 2 and Block 1$\rightarrow$ Block 3.

We first assume the result of *ISZERO(GT(a, num))* is "1" and the path is Block 1 ->Bock 2. In this case, the PC points to the ID 148. The jump type of this path is *conditional jump*. After executing the instructions on ID 148-152, the EVM falls to execute block 4. The jump type from block 2 to block 4 is *fall*. When executing the first instruction of the block 4, the EVM stack holds two values, i.e., *num* and *0*. Block 4 then returns the value 0 to the environment and uses instruction *STOP* to finish the execution.

We then assume the result of *ISZERO(GT(a, num))* is "0" and the path is Block 1 ->Bock 3. In this case, the PC points to the ID 141. The jump type of this path is *fall execution*. *JUMP* refers to an unconditional jump; it reads one value from the top of the stack. The value reads by *JUMP* in ID 147 is '153'. After executing the instructions on ID 141-147, the EVM then jumps to execute block 4. The jump type from block 3 to block 4 is an *unconditional jump*. When executing the first instruction of the block 4, the EVM stack holds two values, i.e., *num* and *1*. Block 4 then returns the value 1 to the environment and uses instruction *STOP* to finish the execution.

When executing a conditional jump, we should determine the satisfiability of the conditional expression, which is typically realized by invoking an SMT (satisfiability modulo theories) solver [22], e.g., Z3 [23]. If the SMT solver cannot find a solution, we consider the corresponding program path as infeasible. Therefore, symbolic execution can be used to discover dead code. However, there may be little dead code in EVM bytecode, because the compiler can eliminate dead code during the compilation of smart contracts. To accelerate our analysis, we consider the conditional expression, which is equal to 0 as unsatisfiable and all other conditional expressions as satisfiable, without checking their satisfiability.

TABLE 2: The Information Required to Detect Each Contract Defect

Contract Defect	Control Flow Information	Symbolic State
Transaction State Dependency (TSD)		✓
DoS Under External Influence (DuEI)	✓	✓
Strict Balance Equality (SBE)	✓	✓
Reentrancy (RE)	✓	✓
Nested Call (NC)	✓	✓
Greedy Contract (GC)	✓	✓
Unchecked External Calls (UEC)		✓
Block Info Dependency (BID)	✓	

3.4 Feature Detector

To detect contract defects at the bytecode level, we need to identify some specific behaviors from their opcodes. In this part, we introduce three features that we use when detecting contract defects.

3.4.1 Money Call

To detect *Reentrancy*, we need to identify whether a smart contract can transfer Ethers to other contracts. Ethereum provides three methods to transfer Ethers, i.e., *address.send()*, *address.transfer()*, *address.call().value()*. All of these three methods generate a *CALL* instruction. However, only detecting the *CALL* instruction is not enough, as many other behaviors can also generate *CALL* instruction, e.g., calling functions on other contracts or library. In this paper, if a *CALL* instruction is generated by functions which are used to transfer Ethers, we call this *CALL* instruction a *Money-CALL*. Otherwise, the *CALL* instruction is a *No-Money-CALL*. *CALL* reads seven values from EVM stack. The first three values represent the gas limitation, recipient address, transfer amounts, respectively. If the transfer amount is larger than 0, the *CALL* instruction is a *Money-CALL*.

However, only detecting *Money-CALL* is still not enough, as *address.send()* and *address.transfer()* will limit the maximum gas consumption to 2300, which is not enough to send Ethers. Therefore, these two methods also cannot cause *Reentrancy*. If the *CALL* instruction is generated by

address.send() and *address.transfer()*, a specific number "2300" will be pushed into EVM stack, which represented the maximum gas consumption. So, if *CALL* instruction reads a specific number "2300" from the EVM stack, the *CALL* instruction is generated by *address.send()* and *address.transfer()*. We call this *CALL* instruction a *Gas-Limited-Money-CALL*. Otherwise, if the first value read by *CALL* instruction does not contain a specific value "2300", we assume that the *CALL* instruction is generated by *address.call().value()*. We call this *CALL* instruction a *Gas-Unlimited-Money-CALL*.

3.4.2 Loop Block

After constructing the CFG, we need to detect which block is the start of a loop and which blocks make up the body of the loop. To detect this information, we first traverse the path of the CFG by utilizing *DFS* (Depth-first-search) [24] and then flag all blocks we visit. If there is a block that has been visited, this block is the start of a loop, and other blocks in this cycle are the loop bodies. Since some smart contracts are very complicated, it may contain a large number of paths. To reduce the computational effort, we use the strategy of pruning. For example, block A is the destination of many other blocks, and we find the path of block A does not contain any cycles. We do not need to visit the remaining paths when other paths encounter block A.

3.4.3 Payable Function

A smart contract can receive Ethers only if it contains payable functions [2]. To detect whether a function is payable or not, we can inspect the first block of each function. *CALLVALUE* instruction is used to get the received Ether amount. If a smart contract receives Ethers, *CALLVALUE* instruction will get a non-zero value. This value can be checked by the *ISZERO* instruction to know whether a transaction contains Ethers. If the function is not payable, when receiving Ethers, it will throw an exception and terminate the execution.

To find the first block, we first rank all instructions by their index ID. All conditional jumps positioned before the first *JUMPDEST* instruction are the start position of each function. EVM uses a hash value to identity functions; when EVM receives a function call, it first compares the received value to each function's hash value. If a function's hash value is equal to the received hash value, it will jump to the destination, which indicates a function's start position. Otherwise, it will fall to *fallback* function, whose start position is the first *JUMPDEST* instruction.

3.5 DefectChecker

Table 2 describes the information required to detect each kind of contract defect. To detect TSD and UEC, *DefectChecker* only needs symbolic states computed by symbolic execution, as we only need to check whether *ORIGIN* and *CALL* instructions are read by *EQ* and *ISZERO* instruction, respectively. *DefectChecker* only needs control flow information to detect BID, as we only need to check whether the conditional expression contains block related instructions, e.g., "BLOCKHASH".

To detect the other 5 contract defects, *DefectChecker* needs both control flow information and symbolic states. In the previous subsection, we introduce three features detected by the feature detector, i.e., *Money Call*, *Loop Block*, and *Payable function*. *Money Call* needs symbolic states, so to detect it, *DefectChecker* needs check the values on the EVM stack. *Loop Block* and *Payable function* require control flow information, as they both need CFGs to locate the loop and the start of the function, respectively. *NC*, *DuEI*, *GC*, and *Reentrancy* all need to detect *Money Call*. *DuEI* and *NC* also need to detect *Loop Block*; *GC* needs to detect *Payable function*. To detect *Reentrancy*, *DefectChecker* needs to travel all the paths that contain the *Gas-Unlimited-Money Call*, which needs the help of the CFG. To detect *SBE*, *DefectChecker* needs to check whether the *BALANCE* instruction is read by the *EQ* instruction in the conditional expressions, which needs both control flow information and symbolic state.

Below we describe the detailed patterns that we use to determine whether a smart contract contains one or more of the contract defects.

3.5.1 Transaction State Dependency

tx.origin generates an *ORIGIN* instruction. We first locate all *ORIGIN* instructions. We then check whether there is an *ORIGIN* that is read by an *EQ* instruction. The *EQ* instruction reads two values from EVM stack and verifies whether these two values are equal. If the contract contains this kind of contract defect *ORIGIN* instruction will compare to an address value. Ethereum uses a 40-bit value to indicate an address, and all addresses conform to the EIP55 standard [25].

3.5.2 DoS Under External Influence

If a smart contract contains this contract defect, there will be a part of the instructions that check the return value of the *Money CALL*, and then terminate the loop. To detect this contract defect, we first find loop-related blocks. Then, we check whether there is a block that contains *Money CALL*, and the type of the block is *conditional*, as it needs to check the return value. Then, this block jumps to a block, which type is *terminal*.

3.5.3 Strict Balance Equality

This kind of contract defect can make a part of the code never be executed. We need to check whether there is a conditional expression that contains the related pattern. *BALANCE* instruction is used to get the balance of a contract. If a *BALANCE* instruction is read by *EQ*, it means there is a strict balance equality check. If this check happens at a conditional jump expression, it means this contract contains this contract defect.

3.5.4 Reentrancy

The *SLOAD* instruction is used to get a value from storage [2]. It reads a value (named *Slot ID*) from the EVM stack and puts the result that reads from storage back onto the EVM stack. Using *listing 4* as an example, *Victim* contracts do not make the balance of an *Attack* contract to zero (L7) before sending Ethers (L6), which allows an *Attack* contract to withdraw Ethers again. To detect this contract defect, we first need to obtain paths that contain *Gas-Unlimited-Money-Call*, because only this kind of *CALL* can cause *Reentrancy*.

We then need to obtain all conditional expressions on these paths. The amount that is sent by the victim contract is usually checked before sending it to attacker contracts, and this amount is loaded from *storage*. In this case, we need to check if the conditional expression contains *SLOAD* instructions and get its *Slot ID*. If this value still holds and does not be updated when executing *CALL* instruction, it means *CALL* instruction can be executed again and cause *Reentrancy*. To check whether the storage value is updated, we need to detect whether the same *Slot ID* that is read by *SLOAD* is written by *SSTORE* instruction. (*SSTORE* instruction is used to save data to memory. It reads two values from EVM stack, i.e., slot id and value that are written to storage.)

3.5.5 Nested Call

Using *listing 5* as an example, array *members* is a storage variable, all of its value, including its length, are stored on storage. To get its length, *SLOAD* instruction reads its *Slot ID* δ from EVM stack, and this value is the position that stores the value of *members.length*. To detect this contract defect, the first step is to find the start block of a loop and get the *Slot ID*. Then, we need to check whether this loop limits its size. If the loop limits its size, the same *Slot ID* δ will be read in the loop body again, and this value will be compared with another value. If a smart contract contains a loop that does not limit its size but contains a *Money-Call*, *Nest Call* is detected in this contract.

3.5.6 Greedy Contract

A smart contract can transfer money through a *Money CALL* or *selfdestruct* function. *selfdestruct* function generates *SELFDESTRUCT* instruction. If a smart contract contains payable functions but does not have either a *Money CALL* or *SELFDESTRUCT* instruction, the contract is a *Greedy Contract*.

3.5.7 Unchecked External Calls

The external call returns a boolean value. If the result is checked by the contract, it will generate an *ISZERO* instruction. To detect this contract defect, we first locate *CALL* instructions. Then, we check whether each *CALL* instruction is read by *ISZERO*. If there is a *CALL* that is not checked by *ISZERO*, this contract defect is detected.

3.5.8 Block Info Dependency

Detecting this contract defect is similar to *Strict Balance Equality*. This contract defect can allow miners to control the contract, as miners can change the value of some block information, which affects the result of the conditional expression. If the conditional expression contains block related instructions, i.e., *"BLOCKHASH"*, *"COINBASE"*, *"NUMBER"*, *"DIFFICULTY"*, *"GASLIMIT"*, it means the contract contains this contract defect.

4 Evaluation

To measure the efficacy of *DefectChecker*, we present results based on applying it to an open-sourced dataset and present our experimental results analysis in this section.

TABLE 3: Some Features of Dataset

Features	Min	Max	Mean	SD
Lines of Code	5	2,239	393.6	356.8
# of Functions	1	174	30.1	621.6
# of Instructions	7	15,355	3,597.3	2,523.7
CC	1	132	30.3	22.4
Ethers	0	1,500,000	7,844.9	1,704,552.7

4.1 Experimental Setup

All experiments were performed on a PC running Mac OS 10.14.4 and equipped with an Intel i7 6-core CPU and 16 GB of memory. We use Solidity 0.4.25 as the compiler to compile source code into bytecode, and use EVM 1.8.14 to disassemble the bytecode to its opcodes.

4.2 Dataset

The dataset we used to evaluate *DefectChecker* was released in our previous work [11]. We first crawled all 17,013 open sourced smart contracts from Etherscan. Then, we randomly selected 600 smart contracts from these contracts. We found 13 smart contracts do not contain any contents. Thus, we removed them from our dataset. Finally, we obtained 587 smart contracts from Etherscan. These contracts have 231,098 lines of the code and more than 4 million Ethers in their balance.

Table 3 shows some key features of the dataset, i.e., lines of code, number of functions in the contracts, number of instructions in the contracts, cyclomatic complexity [26] and Ethers hold by the contracts. Cyclomatic complexity is a software metric that indicates the complexity of a program, and it is computed by analyzing the control flow graph. The formulation to compute it is: $\boldsymbol{E - N + 2P}$. E is the number of edges on CFG; N is the number of nodes on CFG and P is the number of connected components on CFG. Since CFG is a connected graph, so P always equal to 1, and the formulation can be simplified as: $\boldsymbol{E - N + 2}$.

The simplest contract in our dataset only contains one constructor function with 7 instructions and a cyclomatic complexity of 1. The contract with the highest cyclomatic complexity has 11,696 instructions and 2,004 lines of code. The richest contract in our dataset holds 1.5 million Ethers, while the poorest contract has no Ethers in its balance.

Two authors of our previous work manually labeled the dataset. They both have three years of experience working on smart-contract-based development and research, and took part in the process of defining contract defects. Thus, they have a very good understanding of the smart contract programming and contract defects introduced in this paper. They first manually labeled the dataset independently. Then, they discussed the disagreements after completing the labeling process and gave the final results. Their overall Kappa value [27] was 0.71, which shows a substantial agreement between them.

In this work, we developed a tool named *DefectChecker* to detect eight contract defects with severity impact levels 1-3. The numbers of each type of contract defect in our dataset are shown in Table 4. This shows that *Block Info Dependency* is the most frequent contract defect in our dataset, while *Transaction State Dependency* and *Strict Balance Equality* are

TABLE 4: Experimental results for *DefectChecker*.

Defects	#Defects	#TP	#TN	#FP	#FN	P(%)	R(%)	F(%)
TSD	5	5	474	0	0	100.0	100.0	100.0
DuEI	6	6	466	7	0	46.2	100.0	63.2
SBE	5	4	474	0	1	100.0	80.0	88.9
RE	12	10	461	6	2	62.5	83.3	71.4
NC	13	9	464	2	4	81.8	69.2	75.0
GC	6	6	473	0	0	100.0	100.0	100.0
UEC	22	20	454	3	2	87.0	90.9	88.9
BID	42	41	437	0	1	100.0	97.6	98.8

the least popular. Their numbers are 42, 5, and 5, respectively. *DefectChecker* aims at Solidity version 0.4.0+, which is the most widely used version at the time of writing this paper [28]. However, some smart contracts are designed for Solidity version 0.2.0+ and 0.3.0+. Thus, we removed eight smart contracts and used the remaining 579 smart contracts as our ground truth.

Among the six tools we introduced in Table 5, only *Zeus* open sourced their dataset. However, *Zeus* still has four kinds of defects which are not included in their dataset. Also, the *Zeus* authors did not provide the detail of how to built their dataset. Their paper only mentioned that "they manually validated each result without providing any details, e.g., the number of people who labeled the dataset, and whether they are professional smart contract developers or not. Thus, we did not use these datasets.

4.3 Evaluation Methods and Metrics

There are seven measurements obtained from our experiments: True Positive (TP), True Negative (TN), False Positive (FP), False Negative (FN), Precision (P), Recall (R) and F-Measure (F). TP indicates the results which correctly predict a contract defect in a smart contract. TN indicates the results which correctly predict a smart contract does not have a defect. FP and FN indicate the results which incorrectly predict that a smart contract contains and does not contain a contract defect. *Precision*, *Recall*, and *F-Measure* can be calculated as:

$$Precision = \frac{\#TP}{\#TP + \#FP} \times 100\% \quad (1)$$

$$Recall = \frac{\#TP}{\#TP + \#FN} \times 100\% \quad (2)$$

$$F\text{-}Measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \times 100\% \quad (3)$$

4.4 Experimental Results and Analysis

Table 4 summarizes the results of applying *DefectChecker* to our previous work's dataset. The first column is the contract defects that need to be detected. The second column is the number of contract defects in our dataset (ground truth). The remaining seven columns are used to measure the performance of *DefectChecker*. Below, we discuss the analysis of each contract defect.

(1). Transaction State Dependency. *DefectChecker* detects 5 smart contracts containing this contract defect among 579 smart contracts with 0 false positives and negatives.

(2). DoS Under External Influence. *DefectChecker* detects 13 smart contracts that have this contract defect among 579 smart contracts with 7 false positives and 0 negatives. The 7 errors are due to the error identification of a loop.

In our detection method, we first split the bytecode into several blocks. Then, symbolic execution is used to find the edge between blocks. We traverse the path of CFG by using *DFS*. If there is a block that has been visited, we regard this block as the start of the loop (See Section 3.4.2). Since we regard all the paths are reachable, thus we only flag whether two blocks have an edge. This mechanism leads to false positives in detecting loops.

In Listing 10, all the L9, L10, and L11 hold a single block, respectively, and function *sub()* holds several blocks. EVM first executes the block of line 9, then executes the blocks of function *sub()* in line 2. After the execution of blocks of line 10, line 11, respectively, the blocks of function *sub()* will be executed again. Therefore, when traversing the CFG by using DFS, we can find that there is a cycle (fun sub()→L10→L11 →fun sub()). Since we regard all the paths are reachable, we cannot know that the blocks of function *sub()* cannot jump the block of L10, after executing the block of L11.

This kind of false positive can be addressed if we execute the loop continuously. Using a loop "*for (int i = 0; i <100; i++)* as an example; we need to record the state of variable *i*, and check whether the expression *(i <100)* is satisfied or not. If we prove the loop can execute continuously, we can confirm it is a real loop not the error we show in Listing 10. However, we need the assistance of an SMT solver to execute the loop, and executing the loop continuously is also time consuming. Thus, we believe the advantages of removing the use of an SMT solver in our approach outweighs the disadvantages.

```
1  library SafeMath {
2    function sub(uint256 a, uint256 b) internal
         returns (uint256) {
3      assert(b <= a);
4      return a - b;}}
5  contract Mainsale {
6    using SafeMath for uint256;
7    uint256 public total;
8    function() payable {
9      uint amount = total.sub(100);
10     msg.sender.transfer(amount);
11     uint contri = msg.value.sub(amount);}}
```

Listing 10: Error Loop Example

(3). Strict Balance Equality. *DefectChecker* detects 4 smart contracts that contain *Strict Balance Equality* with 0 false positives and 1 false negative. The cause of the error is that the contract defect related to several functions. For example, the contract in listing 11 uses a global variable *balance* to represent the contract's balance. Callers first call function *getBalance* to obtain the balance. The balance will then be checked in Line 5. To detect this contract defect, we need to know that the global variable *balance* represents the contract balance. Therefore, the contract defect can only be detected when we know users will first invoke *getBalance()* and then call *DefectFunction()*. However, it is not easy to detect this

contract defect at the bytecode level, as the two operations (i.e., *balance == 1 eth* and *balance = this.balance*) are in two independent functions, and we do not know the calling sequence.

```
contract Demo{
  uint balance = 0;
  function getBalance(){ balance = this.balance;}
  function DefectFunction() {
    if(balance == 1 eth)
    doSomthing;} }
```

Listing 11: Strict Balance Equality - False Negative Example

(4). Reentrancy. *DefectChecker* detects 16 smart contracts that contain *Reentrancy*, with 6 false positives and 2 false negatives. The false positives are because of error-money-call detection. A smart contract contains *Reentrancy* must have a *Gas-Unlimited-Money-Call*. To detect it, we first need to check whether the gas limits set are larger than 2,300 gas and the transfer amount is larger than 0. However, in some examples, these two values are represented by complicated symbolic expressions. Some expressions also contain values that read from storage (read by *SLOAD*). Thus their specific values can not be determined by static analysis. Therefore, *DefectChecker* failed to detect them. When *DefectChecker* encounters complicated symbolic expressions, the default value is larger than 2,300 gas and larger than 0, this leads to false positives. When detecting this contract defect, we need to check whether the Slot ID read by *SLOAD* instruction still holds when executing *CALL* instruction. Some Slot IDs are also represented by complicated symbolic expressions. *DefectChecker* failed to detect whether they are equal, which leads to reporting false negatives.

When detecting Money-Call, we use *Gas-Limited-Money-Call* as default, if we cannot figure out the exact value of the gas limit symbolically. We also conduct another experiment, which uses Gas-Limited-Money-Call as the default. However, *DefectChecker* failed to detect any *Reentrancy* default. The reason is that the *Gas-Limited-Money-Call* usually is easy to detect, as *address.transfer(), address.send()* will put a specific value "2300" to the EVM stack. Thus, we just need to detect the specific value. However, the gas limit of Gas-Unlimited-Call is not easy to detect, as it usually uses a complicated expression to represent the gas. Since *address.call.value()* will not change the gas cost. In most situations, this method will not lead to an out-of-gas error. This is the reason why we use Gas-Unlimited-Call as our default.

(5). Nested Call. *DefectChecker* detects that 11 smart contracts contain a *Nested Call* defect. Among these 11 smart contracts, we have 2 false positives and 4 false negatives. The cause of the false positives is also the error identification of the loop, which is the same with *DoS Under External Influence*. The false negatives are because of the complicated data structure. When detecting this contract defect, the first step is to know whether the loop iterations are related to the array's length. We use the *SLOAD* instruction related pattern to obtain the loop iterations, as described in Section 3.5.5. However, as shown in Listing 12, *self* is a structure and its length is obtained through an external function. Since external functions can be designed in different ways, it is challenging to design a pattern to detect it.

```
for (uint i; i<self.keys.length; i++) {
    self.data[ self.keys[i]].transfer(1 Ether);}
```

Listing 12: Nest Call - False Negative Example

(6). Greedy Contract. *DefectChecker* detects 6 *Greedy Contracts*, with 0 false positives and negatives.

(7). Unchecked External Call. *DefectChecker* reports 23 contracts have this kind of contract defect, with 3 false positives and 2 false negatives. We analyzed the false positive examples and find that these contracts use the return value of *send()* as function's return value and check the return value in other functions. For example, *addr.send()* as shown in listing 13 is the return value of function *Example*, and the value is checked in the callee programs. The false negatives are because the defect happens in a constructor function, while the bytecode of the constructor function is not contained in runtime bytecode. Therefore, we missed it. However, the contract defects in the constructor function will not harm the deployed contracts, as the constructor function will only be executed once when deploying the contracts to the blockchain.

```
function Example(Address addr) returns (bool) {
    return addr.send();}
```

Listing 13: Unchecked External Call - False Positive Example

(8). Block Info Dependency. *DefectChecker* detects 41 smart contracts contain this contract defects, with 0 false positives and 1 false negative. The cause of the false negative is similar to the one with *Strict Balance Equality*. The defect contract uses a global variable to represent block information and uses this global variable in other functions, which causes the contract defect to be detected.

4.5 Comparison with state-of-the-art tools

In our previous work, we investigated whether there are existing tools that can detect some of the contract defects we have defined. We first collected all the papers from top Security and SE conferences/journals, i.e., CCS, S&P, USENIX Security, NDSS, ACSAC, ASE, FSE, ICSE, TSE, TIFS, and TOSEM from 2016 to 2019. Then, we only retain the papers whose titles have the key words "smart contract", "Ethereum" or "blockchain". After that, we manually read the abstract to verify their relevance. Finally, we found only four papers that are related to smart contract defects, i.e., Oyente [19], Maian [29], Zeus [33], and ContractFuzzer [32].

To enlarge our baseline methods, we use the same method as proposed by Kitchenham et al. [34]. We first read the references of these 4 relevant papers, and tried to find whether there are existing tools that can detect the defined contract defects. If there is a relevant paper, we read its references repeatedly, until no new paper can be found. In this way we also found two other tools, i.e., Securify [30] and Mythril [31].

Table 5 shows the input and contract defects that can be detected by these tools. The last column shows the number of the defects can be detected by these tools except the mentioned 8 contract defects. As we know, the bytecode of smart contract on Ethereum are visible to everyone, but only

TABLE 5: Input and Defects Detected of Each Tool

Tools	Input	TSD	DuEI	SBE	RE	NC	GC	UEC	BID	# of Other Defects
DefectChecker	Bytecode	✓	✓	✓	✓	✓	✓	✓	✓	0
Oyente [19]	Bytecode				✓			✓	✓	1
Maian [29]	Bytecode						✓			2
Securify [30]	Bytecode				✓			✓		7
Mythril [31]	Bytecode	✓	✓	✓	✓	✓		✓		28
Contractfuzzer [32]	Bytecode + ABI				✓			✓	✓	3
Zeus [33]	Source Code	✓			✓			✓	✓	3

TABLE 6: Experiment result of *Oyente*.

Defects	#Defects	#TP	#TN	#FP	#FN	P(%)	R(%)	F(%)
RE	12	2	94	373	10	2.1	16.7	3.7
UEC	22	16	448	9	6	64.0	72.7	68.1
BID	42	11	431	6	31	64.7	26.2	37.3

TABLE 7: Experiment result of *Mythril*.

Defects	#Defects	#TP	#TN	#FP	#FN	P(%)	R(%)	F(%)
TSD	5	0	474	0	5	0	0	0
DuEI	6	1	245	228	5	0.4	16.7	0.8
SBE	5	0	474	0	5	0	0	0
RE	12	5	280	187	7	2.6	41.7	4.9
NC	13	2	414	52	11	3.7	15.4	6.0
UEC	22	11	436	21	11	34.4	50.0	40.8

less than 1% of the smart contracts open up their source code [20]. Therefore, detecting contract defects from the bytecode level is very important. To make the comparison fair, we select *Oyente, MAIAN, Securify* and *Mythril* as our baseline tools, since they can detect contract defects at the bytecode level, the same as *DefectChecker*. However, we found that *Maian* has not been updated to support the latest Ethereum environment and so we could not run *MAIAN* on our dataset. For example, they use methods provided by *web3* [35] to obtain contracts' information on Ethereum. However, the methods they used have been removed and did not support the current version of Ethereum that we used. In addition, *DefectChecker* gets 100% F-Measure when detecting *Greedy Contract*. In this case, we do not compare with *MAIAN*, and choose *Oyente, Securify* and *Mythril* as our baseline tools.

Oyente detects three kinds of security-related vulnerabilities for smart contracts. These three kinds of security-related vulnerabilities are the same as our *Unchecked External Calls*, *Block Info Dependency* and *Reentrancy*. *Mythril* [31] is a tool developed by *ConsenSys*, which is a leading global blockchain technology company. They find security problems from online posts or news, which is similar to our previous work [11]. Our previous work analyzed the posts from StackExchange posts and defined 20 contract defects. *Mythril* can detect 6 contract defects as shown in Table 7. *Securify* is a smart contract security analyzer that takes EVM bytecode as input. It first decompiles EVM bytecode and analyzes the semantic facts of the decompiled code. In our study, *Securify* uses several security patterns to detect related vulnerabilities. *Securify* can detect *Reentrancy* and *Unchecked External Call*, which can also be detected by *DefectChecker*.

Table 6 shows the results of running *Oyente* on our previous dataset [11]. The F-score of *Oyente* in detecting RE, UEC, and BID are 3.7%, 68.1%, and 37.3%, respectively, while the numbers for DefectChecker are 71.4%, 88.9%, and 98.8%, respectively. We found that *Oyente* only considers *BLOCKHASH* instructions when detecting *Block Info Dependency*, while there are many other instructions, e.g. NUMBER (NUMBER instruction is used to get block's number), that can lead to this contract defect. Besides, *Oyente* also has many false positives when detecting *Reentrancy*. The reason is that they do not distinguish between *send()*, *transfer()* and *call()* functions at the bytecode level, while *send()* and *transfer()* will limit gas to 2300 unit, which cannot cause *Reentrancy*. In addition, the most important reason for these errors is *code coverage*. Code coverage means the percentage of instructions executed. The average code coverage for *Oyente* is 18.9%, while the number for *DefectChecker* is 77.1%. Low code coverage means only a small part of the code can be analyzed for contract defect occurrence, which can lead to a large number of false positives and negatives. There are three reasons that lead to the low coverage of Oyente compared to DefectChecker. First, *Oyente* checks whether a path can be reached, while *DefectChecker* assumes that all the paths are reachable. *Oyente* also only optimizes for Solidity Version 0.4.19, but there is a wide version coverage in our dataset. Finally, the jump positions of some unconditional jump might not be easy to find. To be specific, the jump position might be a result of a complicated expression. Thus both *Oyente* and *DefectChecker* can fail to detect these unconditional jumps, and it is the reason why DefectChecker misses some blocks.

Table 7 shows the results of *Mythril*. *Mythril* fails to detect *Transaction State Dependency* and *Strict Balance Equality* in our dataset. In addition, its results contain many false positives, especially in detecting *Reentrancy* and *DoS Under External Influence*. We found that *Mythril* is similar to *Oyente* - it fails to distinguish between *call()* with *transfer()* and *send()*, which will not lead to *Reentrancy*. Besides, *Mythril* failed to distinguish loop related patterns, which lead to errors when detecting loop related defects, e.g., *DoS Under External Influence* or *Nest Call*.

Table 8 presents the results of *Securify*. *Securify* can detect two common defects with *DefectChecker*, i.e., *Reentrancy* and *Unchecked External Call*. All the *DefectChecker*, *Oyente*, *Mythril*, and *Securify* can detect these two defects. The performance of *Securify* in testing *Reentrancy* (4.9%) is better than *Oyente* (3.7%), and similar to *Mythril* (4.9%), but much worse than *DefectChecker* (71.4%). In terms of detecting *Unchecked External Call*, the F-score of *Securify* (62.5%) is a little bit worse than *Oyente* (68.1%) and much better than *Mythril*. *DefectChecker* still get the best F-score, which receives 88.9% in detecting *Unchecked External Call*

To compare the results between all four tools, we add a comparison of F-measure in Table 9, which shows that

TABLE 8: Experiment result of *Securify*.

Defects	#Defects	#TP	#TN	#FP	#FN	P(%)	R(%)	F(%)
RE	12	1	439	28	11	3.5	8.3	4.9
UEC	22	10	457	0	12	100.0	45.5	62.5

TABLE 9: Result Comparison(F-Measure) between Four Tools

Tools	TSD	DuEI	SBE	RE	NC	GC	UEC	BID
DefectChecker	**100.0%**	**63.2%**	**88.9%**	**71.4%**	**75.0%**	**100.0%**	**88.9%**	**98.8%**
Oyente	/	/	/	3.7%	/	/	68.1%	37.3%
Securify	/	/	/	4.9%	/	/	62.5%	/
Mythril	0%	0.8%	0%	4.9%	6.0%	/	40.8%	/

DefectChecker obtains the best F-measure of all four tools.

TABLE 10: Overall Precision, Recall, and F-Measure of Each Tool

Tools	O. P. (%)	O. R (%)	O. F. (%)
DefectChecker	88.3	90.9	88.8
Oyente	54.6	38.2	40.9
Securify	65.9	32.4	42.2
Mythril	13.3	30.2	16.5

TABLE 11: Time Consumption of Each Tool

Tools	Avg.	Max	Min	S.D.
DefectChecker	0.15s	2.42s	0.04s	5.43
Oyente	18.48s	1,096.32s	0.28s	2,877.64
Securify	21.55s	1,203.99s	0.37s	3,384.39
Mythril	103.55s	2,480.26s	1.58s	13,063.80

We also calculate the overall precision, recall, and F-measure of all four tools on the whole experimental dataset. Using overall-precision as the example, the overall result is calculated by $\frac{\sum_{i=1}^{n} p_{c_i} \times |c_i|}{\sum_{i=1}^{n} |c_i|}$, in which p_{c_i} is the precision of the contract defect i, $|c_i|$ is the number of contract defect i in the whole dataset. The results are given in Table 10, which clearly shows that *DefectChecker* obtains the best results in detecting contract defects.

Time Consumption. We calculate the time to analyze one smart contract to evaluate each tool. To make the evaluation accurate, we kill all the background processes in our machine when testing the tool to ensure the environment is clean. For each tool, we run it for 10 times and record the average time to test one smart contract in our dataset.

Table 11 shows the time consumption results of each tool. The second column of the table gives the average time consumption to test a smart contract for each tool. The speed of *DefectChecker* is the fastest in these four tools. It only needs 0.15s to analyze one smart contract. *Oyente* and *Securify* have similar running times. *Oyente* needs 18.48s to analyze one smart contract, and the time for *Securify* is 21.55s. *Mythril* is the slowest tool; it needs 103.55s to analyze one smart contract. The maximum time to analyze a smart contract of *DefectChecker* is 2.42s, while the time for *Oyente*, *Securify*, and *Mythril* are 1096.32s, 1203.99s and 2480.26s, respectively. The simplest smart contract in our dataset only contains 7 lines with a single constructor function. *DefectChecker* needs 0.04s to analyze it, while the time for *Oyente*, *Securify*, and *Mythril* are 0.28s, 0.37s and 1.58s, respectively. *DefectChecker* also has the smallest Standard Deviation value among these four tools, which shows that *DefectChecker* has the most stable speed in analyzing a smart contract.

In conclusion, the efficiency of these four tools is in order: *DefectChecker* >*Oyente* >*Securify* >*Mythril*.

4.6 Threats to Validity

Internal Validity. We used a dataset released in our previous work [11] as the ground truth to evaluate *DefectChecker*. Since the people who developed *DefectChecker* are the same as the people who labeled the dataset, it is likely that their familiarity with the dataset might lead to potential optimization or omissions when developing *DefectChecker*. We tried to use the datasets of the baseline tools to evaluate *DefectChecker*. However, we failed to find the dataset. Luu et al. run *Oyente* on 19,366 contracts. They only manually check the correctness of some examples, instead of using a complete dataset to evaluate *Oyente*. We can only find some false positive and true positive values on their paper. *Securify* uses a complete dataset which consists of 100 smart contracts. However, they do not open their dataset to the public. *Mythril* is a tool from industry. They even do not have an evaluation section in their technical papers. Thus, we had to build our own dataset. To reduce the influence of our dataset, we first wrote a few demo smart contracts when developing *DefectChecker* and used these to conduct small-scale testing of our proposed tool. Then, we conducted large-scale testing by using real world bytecode we crawled from the Ethereum blockchain. The dataset is the same as that we introduced in Section 5. During this large-scale testing, we randomly choose a set of smart contracts that can find their source code. We use these smart contracts to improve the performance and patterns that are used to detect contract defects. We admit that the familiarity with the ground truth dataset might lead to a bias, but the methods we used to develop *DefectChecker* can reduce this influence.

External Validity. The dataset we used to evaluate *DefectChecker* is based on manual analysis, which may contain false positives and negatives. To address this problem, we double-checked the results and used them to update the dataset when we found some mistakes. Another threat is that Solidity is a fast-growing programming language. There are nine versions released in 2018, which may add or modify any features of the previous version. *DefectChecker* is designed based on Solidity version 0.4.0+, which is the most popular version in the time of writing the paper [28]. In the future, more smart contracts may use higher versions, which may make our tool unable to work.

5 A Large Scale Evaluation

In the previous section, we showed that *DefectChecker* has an excellent performance when applied to a small scale dataset. In this section, to validate *DefectChecker* is still usable to find contract defects in real-world smart contracts, we ran *DefectChecker* on a large scale dataset that we crawled from Ethereum blockchain, and show the contract defects as

TABLE 12: Contract Defects in Ethereum

Contract Defects	# Defects	# Percentage
Transaction State Dependency	1,669	1.0%
DoS Under External Influence	2,116	1.3%
Strict Balance Equality	390	0.2%
Reentrancy	3,892	2.4%
Nested Call	1,043	0.6%
Greedy Contract	3,139	1.9%
Unchecked External Calls	12,439	7.5%
Block Info Dependency	5,201	3.1%

found by *DefectChecker*. We give two real-world attacks as case studies to show how harmful these contract defects are.

5.1 Dataset

To identify whether contract defects are actually prevalent in a large-scale, real-world dataset, we crawled bytecode from Ethereum blockchain by 2019.01 and obtained 183,706 distinct bytecode. Since some smart contract versions are not supported by *DefectChecker*, and so we removed them from our experimental dataset. Finally, we ran *DefectChecker* on 165,621 distinct smart contract bytecode. All these bytecode are runtime bytecode. Runtime bytecode does not contain information on their constructor function. It is the default bytecode stored on the Ethereum.

5.2 Contract Defects on Ethereum

We ran *DefectChecker* on 165,621 smart contract bytecode. The detailed results are given in Table 12, which aims to show the frequency of each defect on Ethereum. Since *DefectChecker* only identifies whether a contract contains a defect or not, if the same kind of defects appears multiple times in a smart contract, we only count it one time in the Table 12. The second column of the table shows how many contracts contain related defects, and the last column gives the percentage of how many contracts contain the defect. If a contract contains multiple defects, all of the defects are counted.

Unchecked External Calls is the most frequent contract defect in the Ethereum, and about 7.5% of real world smart contracts contain this defect. There are about 3.1% of smart contracts that contain *Block Info Dependency*, which is the second most popular contract defect on the blockchain. *Strict Balance Equality* is the rarest of our contract defects. *DefectChecker* only detects 390 smart contracts that have this contract defect. The percentage of *Nested Call* is also less than 1%, with 1,043 (0.6%) smart contracts having this kind of contract defect. The percentage of *Transaction State Dependency* and *DoS Under External Influence* are similar on Ethereum, at about 1.0% and 1.3%, respectively. There are 3,139 greedy contracts on the Ethereum, and 3,892 smart contracts containing the *Reentrancy* problem, which can lead to serious security problems.

We found that there are 16 smart contracts that contain 4 kinds of contract defects, which are thus the most defective contracts. The number of smart contracts that contain 3 kinds of contract defects is 539, and 3,520 smart contracts contain 2 kinds of contract defects. About 25,815 smart contracts contain at least one kind of defect, which means

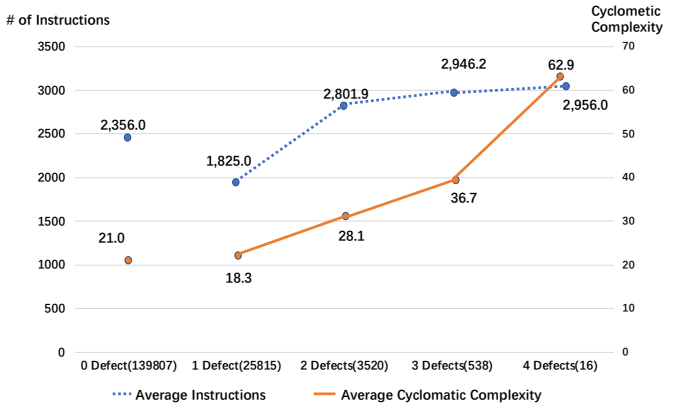

Fig. 3: The relationship between the number of contract defects and number of Instructions & Cyclomatic Complexity

that about 15.9% smart contracts on Ethereum contain some kinds of defects, as reported by our *DefectChecker*.

We utilized cyclomatic complexity [26] and the number of instructions to conduct a further analysis. We computed the cyclomatic complexity and number of instructions for contracts in our dataset. We found that the average cyclomatic complexity of smart contracts in Ethereum is 21.3, and the average number of instructions are 2,342.6. Figure 3 shows the relationship between the number of the contract defects that contained in smart contracts and the number of instructions & cyclomatic complexity. The x-axis means the number of contract defects in a smart contract. The left y-axis is the number of instructions, and the right y-axis is the number of cyclomatic complexity. The two lines have a similar trend.

The number of instructions is proportional to the length of a contracts' code, which can show the contracts' complexity at the code level. The number of cyclomatic complexity indicated the complexity of a program. We performed a linear regression [36] to analyze the relationship between the number of defects with instructions, and the number of defects with cyclomatic complexity. They follow the trend: $y_1 = 353x + 1748$, and $y_2 = 14x + 0.92$, respectively. y_1 is the number of instructions, y_2 is the cyclomatic complexity, and x ($x = [1, 2, 3, 4]$) is the number of defects in a smart contract. The Multiple R values of them are 0.84 and 0.96, respectively. Both of them are larger than 0.8, which shows a positive correlation [36]. Thus, our results show that the more complexity of a contract, the higher probability that it contains smart contract defects.

5.3 Case Study

DefectChecker found some real-world attacks / financial loss from our large-scale testing on the full Ethereum dataset. In this subsection, we give two examples to show the importance of detecting such contract defects.

Case Study 1: The first example is shown in Listing 14. There are 2,335.8 Ethers in the contract balance, and it is worth $552,720 by Mar. 2020. Unfortunately, all the Ethers are locked because of the contract defect, i.e., *Nested Call*. The buggy function in Listing 14 is named *sendReward()*. We highlight two lines of the code (Line 2 and Line 14), which

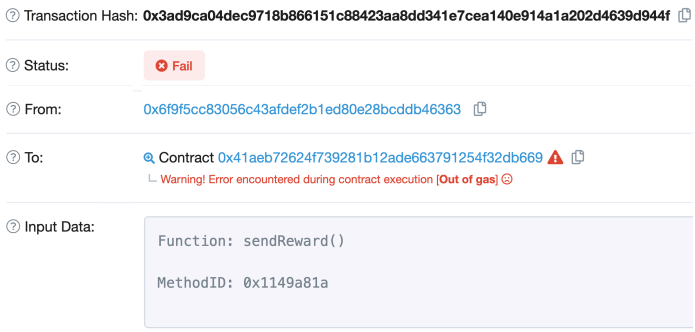

Fig. 4: Transaction Detail of Case Study 1

are related to two contract defects, i.e., *Nested Call* and *DoS Under External Influence*.

There is a loop in the function *sendReward()*, and the loop iterations are increased with the length of *investors[]*. However, the contract does not limit its loop iterations. As we know, sending Ethers is expensive as it needs a large amount of gas consumption, and the contract sends Ethers to the contract users in Line 14. So, the gas consumption of executing *sendReward()* will increase in the length of *investors[]*. When we check the transaction of the contract, we can find that the contract can work normally at first, as the total gas consumption of *sendReward()* does not exceed its maximum gas limitation at that time. However, with the increase of the length of *investors[]*, the total gas cost increases rapidly. The gas cost then eventually exceeds the gas limitation, and leads to an out of gas error. Even worse, since the length of *investors[]* cannot be reduced, once the error happens, the *sendReward()* cannot be called anymore, which means all the Ethers in the balance are locked forever. Figure 4 shows the detail of a failed transaction. It is clear that when a user calls *sendReward()*, the out-of-gas error happens.

```
1  function sendReward() public isOwner{
2    for (uint i = 0; i < investors.length; i ++){
3      address _add = investors[i];
4      User memory _user = addressToUser[_add];
5      if (_user.gameOver){
6          autoReInvest(_add);
7          _user.rebirth = now - (oneLoop / 2);
8          addressToUser[_add] = _user;
9      }else {
10         if (SafeMath.sub(now , _user.rebirth) >=
               oneLoop){
11             address payable needPay = address(
                   uint160(_add));
12             uint staticAmount = getStatic(_add);
13             if (staticAmount > 0){
14                 needPay.transfer(staticAmount);
15             }
16     ...
17     }
18 }
```

Listing 14: Case Study 1 - Contract with Nested Call. Code from Contract: 0x41AeB72624f739281b12aDE663791254F32DB669.

It should be noticed that although the financial loss in the real world example is caused by *Nested Call*, the contract shown in Listing 14 also has another contract defect, namely *DoS Under External Influence*. This contract defect can also lead to the lock of Ethers. Specifically, if the *needPay* (Line 14) is a contract address, the maximum *Gas Limit* will be restricted to 2300 gas units, which is not enough to transfer Ethers. Thus, an *out-of-gas error* will happen in Line 14, and the Ether transfer cannot succeed.

Case Study 2: A second example is a bank contract, which is shown in Listing 15. Users can send Ethers to the *Deposit()* function, and withdraw its Ethers by calling the *CashOut()* function. First, the contract sends Ethers on Line 11 and then reduce the caller's balance on Line 12. However, it can lead to the *Reentrancy* if the caller is an attacking contract. When the victim contract sends Ethers to the attack contract. The fallback function of the attack contract can recall the *CashOut()* function, and steal Ethers of the victim contract. Then, all of the balance in the contract was stolen by the attackers.

Figure 5 shows an attacking transaction which was launched by an attacking contract. The address of the attacking contract starts with 0xdefbe, and the address of the victim contract starts with 0xbabfe. The attack happens three times on block 4919015, 4919567, and 4919662, respectively. First, the attacking contract sent 1 Ether to the victim contract. Then, the victim contract returned back Ethers to the attack contract. From these 3 attacks, the attacking contract stole about 5 Ethers from the victim contract, which were worth about $1,200 at the time of writing the paper. We only show one example in Figure 5. Actually, the victim contract was attacked by multiple attacking contracts, so the financial loss was far more than 5 Ethers.

```
1  function Deposit() public payable{
2    if(msg.value >= MinDeposit){
3      balances[msg.sender]+=msg.value;
4      TransferLog.AddMessage(msg.sender,msg.value,"
           Deposit");
5    }
6  }
7
8  function CashOut(uint _am)
9  {
10   if(_am<=balances[msg.sender]){
11     if(msg.sender.call.value(_am)()){
12         balances[msg.sender]-=_am;
13         TransferLog.AddMessage(msg.sender,_am,"
               CashOut");
14     }
15   }
16 }
```

Listing 15: Case Study 2 - Contract with Reentrancy. Code from Contract: 0xbABfE0AE175b847543724c386700065137d30e3B.

5.4 Threats to Validity

Internal Validity. The dataset we used was crawled from Ethereum, which contains different Solidity versions. *DefectChecker* only supports versions higher than 0.4.0+, and about 20,000 contracts had to be removed from our dataset, which may influence the overall results. However, the bytecode we removed is from many years ago, since the first version of 0.4.0+ was released on Sept. 2016. Even though there are many contract defects in the removed bytecode, these do not represent current smart contract usage.

Another key threat is that we used our *DefectChecker* to get the results, but *DefectChecker* also reports false positives

Block	From	To	Value
4919850	0xbabfe0ae175b84...	0xdefbe144c325d3...	2.000000000002421899 Ether
4919662	0xdefbe144c325d3...	0xbabfe0ae175b84...	1 Ether
4919567	0xbabfe0ae175b84...	0xdefbe144c325d3...	2.000000000002421899 Ether
4919567	0xbabfe0ae175b84...	0xdefbe144c325d3...	1 wei
4919567	0xbabfe0ae175b84...	0xdefbe144c325d3...	1 Ether
4919567	0xdefbe144c325d3...	0xbabfe0ae175b84...	1 Ether
4919015	0xbabfe0ae175b84...	0xdefbe144c325d3...	2.000000000004843799 Ether
4919015	0xbabfe0ae175b84...	0xdefbe144c325d3...	1 wei
4919015	0xbabfe0ae175b84...	0xdefbe144c325d3...	1 Ether
4919015	0xdefbe144c325d3...	0xbabfe0ae175b84...	1 Ether

Fig. 5: Transaction Lists of Case Study 2

and negatives, as shown in the previous section. However, *DefectChecker* is the most accurate and efficient tool that detects contract defects in the bytecode level, as we also demonstrated in the previous section. Therefore, we believe the results and our conclusions from it are reasonable.

External Validity. There are more than 1,000 smart contracts being deployed to Ethereum every day [37]. Many guidance and security detection tools [29], [38] are released to the public, which can help to improve the quality of smart contracts. In this case, the contract defects in smart contracts may decrease, which may lead to different results to what we found and reported in this section.

6 Related Work

Contract Defects on Smart Contracts. Our previous work [11] is the first work that defines 20 smart contract defects on Ethereum by analyzing the post on StackExchange [13]. We first crawl all 17,128 Stack Exchange posts by the time of writing the paper and use key words to filter solidity related posts. After getting Solidity related posts, two authors of the paper use *Open Card Sorting* to find 20 contract defects and divide them into five categories, i.e., *security, availability, performance, maintainability*, and *reusability defects*. According to their paper, although previous works define several security defects, they did not consider the practitioners' perspective. Therefore, we first designed an online survey to collect feedback from developers to validate whether the developers regard the contract defects are harmful. This feedback showed that all the defined contract defects are harmful to smart contracts. We assigned five impact levels to the defined 20 contract defects according to our survey results and the symptoms of the defects. According to our definition, contract defects with impact level 1-3 can lead to unwanted behaviors of contract, e.g., a contract being controlled by attackers.

Smart Contract Security Problems and Detection Tools. Luu et al. [19] introduced four security issues in their work, i.e., mishandled exception, transaction-ordering dependence, timestamp dependence, and reentrancy attack. They proposed a tool named *Oyente*, which is the first symbolic execution based bug detection tool for smart contracts. They first split the bytecode into several blocks, and built a skeletal control flow graph for the detected contract. Then, they utilized Z3 [23] as their SMT solver and symbolically executed each instruction to obtain the full control flow graph. Finally, they designed different patterns to detect whether the input contracts contain the defined security problems. Oyente measured 19,366 existing Ethereum contracts and found 8,519 of them contain the defined security problems.

Kalra et al. [33] developed a tool named Zeus. The tool feeds source code as input and translates them to LLVM bytecode. Zeus can detect seven kinds of security problems (four of them are the same with Oyente), and the other three problems are *unchecked send, Failed send, Integer overflow/underflow*. They also compared their result to Oyente and found Oyente contains many false positives and false negatives. Zeus crawled 1,524 distinct smart contracts from Etherscan [28], Etherchain [39] and EtherCamp [40] explorers to evaluate their tool. The result illustrates that about 94.6% of contracts contain at least one security problem. However, the needs of source code limited their usage.

Jiang et al. [32] proposed a tool named *ContractFuzzer* to test seven security issues. *ContractFuzzer* is the first tool that utilizes fuzzing technology to detect security problems on smart contracts. They tested 6,991 smart contracts and found that 459 of them have issues. However, only less than 0.5% of smart contracts open their ABI to investigate on Ethereum [28], while their tool needs smart contract ABI or source code to generate test case, which limited their usage. In addition, our dataset consisted of 579 bytecode smart contracts, which are not supported by *ContractFuzzer*.

Nikolic et al. [29] developed a tool named MAIAN, which contains two major parts: symbolic analysis and concrete validation. Similar to Oyente, MAIAN utilizes symbolic execution and defines several execution rules to detect these security issues. Their tool takes input data as either bytecode or source code. MAIAN has a different concern compared to our tool. They focus on security issues that can lead to a contract not able to release Ethers, can transfer Ethers to arbitrary addresses, or can be killed by anybody. Their results were deduced from 970,898 smart contracts and they found that a total of 34,200 (2,365 distinct) contracts contain at least one of these three security issues.

ConsenSys is a leading blockchain technology company. They built a website named SWC Registry [41] (Smart Contract Weakness Classification and Test Cases) to collect smart contract security problems from both online posts and news through crowdsourcing. *Mythril* [31] is a tool to detect security problems on this SWC Registy, and their first version was released in May 2018. The method used by *Mythril* is similar to *Oyente*. It first builds a CFG and utilizes Z3 [23] as an SMT solver. Then, it designs several rules to detect related problems. *Mythril* is a tool developed by industry; their instruction manual does not contain any evaluation section on the tool.

Securify [30] is a tool released by Tsankov et al. *Securify* is the first tool that utilizes semantic information to detect security problems on smart contracts. It first decompiles EVM bytecode to and analyzes the semantic facts, including data flow and control flow dependencies. Finally, it checks several security patterns that are written in a specialized

domain-specific language to detect related security problems. *Securify* focuses on two kinds of security problems, i.e., *Stealing Ether* and *Frozen Funds*. There are 9 security issues can that be detected by *Securify*. Tsankov et al. evaluate their tool based on two datasets. First, a large-scale evaluation based on 24,594 smart contracts. Their results show that more than 70% of smart contracts contain at least one of the security problems. Then, they use a small-scale evaluation based on 100 smart contracts to evaluate their proposed tool's effectiveness. To simplify manual inspection, all of these 100 smart contracts are up to 200 lines of code. According to their paper, *Securify* can find more security violations compared to *Oyente* and *Mythril*.

In this paper, we propose a tool named *DefectChecker*, which is the most accurate and the fastest symbolic execution model of smart contract defect detection tool. *DefectChecker* can detect contract defects by analyzing bytecode, while *Zeus* and *ContractFuzzer* need source code and contract ABI, respectively. The bytecode of smart contracts are visible to everyone, while only 1% of smart contracts open up their source code and ABI for the public [20], which restricts their usage. *MAIAN* uses a dynamic analysis method to detect security problems, which is different from our static analysis method. However, we find their tool can not support the current version of Ethereum that we used. *Oyente, Mythril*, and *Securify* use symbolic execution to detect security problems, which are similar to *DefectChecker*, but *DefectChecker* uses *Stack Event* and *Feature Detector* to instead the usage of SMT solver, which makes *DefectChecker* requires less runtime and yet is more accurate than these tools.

Oyente, Mythril, and *Securify* can detect other contract defects that are not supported by *DefectChecker*. Especially for *Mythril*, which can detect 34 kinds of contract defects. We admit that some tools can detect more contract defects than *DefectChecker*, but it is not the main motivation of this paper. Previous works, e.g., *Oyente, Securify*, only proposed several security defects of smart contracts without validating they are really harmful. This is not beneficial for the development of the smart contract ecosystem. In our previous work, we validated whether smart contract developers consider the contract defects we found from StackExchange posts are harmful by using an online survey. In this paper, we proposed *DefectChecker*, which aims to automatically detect the validated contract defects. We use *Oyente, Mythril,* and *Securify* as baseline methods with the aim to show the method we use is more accurate and efficient than these state-of-the-art tools.

Our *DefectChecker* is extensible. As shown in Figure 1, there are three components of *DefectChecker*, i.e., *CFG Builder*, *Feature Detector*, and *Defect Identifier*. *Defect Identifier* uses eight different rules to identify the contract defects, while the other two components can also be used to detect other defects. When detecting other defects, we can define new rules that use the data provided by our Feature Detector, CFG, and Stack Event components. There are many tools built based on the top of Oyente. For example, our previous work *GasChecker* [42] is a tool to detect gas-inefficient Smart Contracts. The tool uses the CFG generated by *Oyente* to detect related gas-inefficient issues. *DefectChecker* has higher efficiency in generating CFG compared to *Oyente*. *GasChecker* can also use the CFG generated by *DefectChecker*. Thus, *DefectChecker* is also extensible to detect other kinds of issues.

7 Conclusion and Future Work

In this paper, we proposed *DefectChecker*, which utilizes symbolic execution to detect smart contract defects by analyzing the contracts' bytecode. *DefectChecker* uses different rules to detect 8 contract defects and achieves a very good result when running on our previous work's dataset. The scores for our tool are much higher than those of the state of the art work e.g. (*Oyente, Mythril*, and *Securify*). We also crawled 165,621 distinct bytecode smart contracts from Ethereum and ran *DefectChecker* on these. Our results show that about 15.89% of smart contracts on Ethereum contain at least one instance of our 8 identified kinds of contract defects.

Two groups can benefit from this work. For smart contract developers, they can utilize *DefectChecker* to check their smart contracts and make them more robust. As *DefectChecker* can detect contract defects from bytecode without the need for source code, developers can utilize *DefectChecker* to check whether the smart contracts they call are secure or not, even if the callee contracts are not open sourced. This can also make their contracts safer. For software engineering researchers, *DefectChecker* provides a good framework to help them solve other smart-contract-related research problems as the CFG generated by *DefectChecker* can be used for other purposes.

DefectChecker has some false positives / negatives when detecting defects, e.g., *NC, DuEI*. As we described in the Section 4.4, adding a SMT Solver can reduce some error cases, while it can increase the time consumption for analyzing a contract. Researchers can conduct future work to combine the method used by *DefectChecker* and a SMT solver, which can balance both efficiency and accuracy. Specifically, researchers can identify which kinds of code patterns can lead to the errors of *DefectChecker*. For example, *DefectChecker* has some false positives in detecting loops. Thus, researchers can use a SMT solver to detect this kind of code to increase the correctness.

Acknowledgements

This research was partially supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE200100021), ARC Laureate Fellowship funding scheme (FL190100035), ARC Discovery grant (DP200100020), National Natural Science Foundation of China (61872057), National Key RD Program of China (2018YFB0804100), and the National Research Foundation, Singapore under its Industry Alignment Fund Prepositioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Working Paper*, 2008.

[2] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Project Yellow Paper*, 2014.

[3] (Apr., 2019) Decentralized Application. [Online]. Available: https://en.wikipedia.org/wiki/Decentralized_application

[4] (Feb., 2019) Cryptokitties. [Online]. Available: https://www.cryptokitties.co/

[5] (Apr., 2019) ICO Ethereum. [Online]. Available: https://etherscan.io/directory/ICOs

[6] (Mar., 2018) Solidity Document. [Online]. Available: http://solidity.readthedocs.io

[7] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang, "An Adaptive Gas Cost Mechanism for Ethereum to Defend Against Under-Priced DoS Attacks," in *International Conference on Information Security Practice and Experience*. Springer, 2017, pp. 3–24.

[8] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhange, "Understanding Ethereum via Graph Analysis," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 1484–1492.

[9] T. Chen, Z. Li, Y. Zhu, J. Chen, X. Luo, J. C.-S. Lui, X. Lin, and X. Zhang, "Understanding Ethereum via Graph Analysis," *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 2, pp. 1–32, 2020.

[10] ISO, "ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary," ISO/IEC/IEEE 24765: 2017 (E), Tech. Rep., 2017.

[11] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining Smart Contract Defects on Ethereum." IEEE Transactions on Software Engineering, 2019.

[12] R. Chillarege *et al.*, "Orthogonal defect classification," *Handbook of Software Reliability Engineering*, pp. 359–399, 1996.

[13] (Jan., 2018) StackExchange. [Online]. Available: https://ethereum.stackexchange.com/

[14] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang, "Towards saving money in using smart contracts," in *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. IEEE, 2018, pp. 81–84.

[15] Ethereum Foundation, "Ethereums white paper." *https://github.com/ethereum/wiki/wiki/White-Paper*, 2014.

[16] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[17] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: thorough, declarative decompilation of smart contracts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1176–1186.

[18] T. Chen, Z. Li, Y. Zhang, X. Luo, T. Wang, T. Hu, X. Xiao, D. Wang, J. Huang, and X. Zhang, "A large-scale empirical study on control flow identification of smart contracts," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–11.

[19] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 254–269.

[20] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "TokenScope: Automatically Detecting Inconsistent Behaviors of currency Tokens in Ethereum," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1503–1520.

[21] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang, "Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts," *IEEE Transactions on Emerging Topics in Computing*, 2020.

[22] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[23] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[24] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[25] (Jan., 2016) EIP-55. [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-55.md

[26] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[27] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[28] (Mar., 2018) Etherscan. [Online]. Available: https://etherscan.io/

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

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

[31] (Aug., 2019) Mythril: Security analysis tool for EVM bytecode. . [Online]. Available: https://github.com/ConsenSys/mythril

[32] 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.

[33] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *25th Annual Network and Distributed System Security Symposium (NDSS18)*, 2018.

[34] B. Kitchenham, "Procedures for performing systematic reviews," *Keele, UK, Keele University*, vol. 33, no. 2004, pp. 1–26, 2004.

[35] (April., 2019) Web3.py. [Online]. Available: https://web3py.readthedocs.io/en/stable/

[36] G. A. Seber and A. J. Lee, *Linear regression analysis*. John Wiley & Sons, 2012, vol. 329.

[37] T. Chen, Z. Li, Y. Zhang, X. Luo, A. Chen, K. Yang, B. Hu, T. Zhu, S. Deng, T. Hu *et al.*, "Dataether: Data Exploration Framework for Ethereum," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1369–1380.

[38] (Mar., 2018) Oyente: An Analysis Tool for Smart Contracts. [Online]. Available: https://github.com/melonproject/oyente

[39] (Mar., 2018) Etherchain. [Online]. Available: https://www.etherchain.org/contracts/

[40] (Mar., 2018) Ethercamp. [Online]. Available: https://live.ether.camp/

[41] (July., 2019) SWC Registry: Smart Contract Weakness Classification and Test Cases. [Online]. Available: https://smartcontractsecurity.github.io/SWC-registry/

[42] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang, "GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts," *IEEE Transactions on Emerging Topics in Computing*, 2020.