HuangGai: Automatic Bug Injection for Ethereum Smart Contracts

Pengcheng Zhang, Feng Xiao, Xiapu Luo, and Hai Dong

Abstract—Although many tools have been developed to detect bugs in smart contracts, the evaluation of these analysis tools has been hindered by the lack of adequate buggy *real contracts* (i.e., smart contracts deployed on Ethereum). This problem prevents conducting reliable performance assessments on the analysis tools. An effective way to solve this problem is to inject bugs into *real contracts* and automatically label the locations and types of the injected bugs. *SolidiFI*, as the first and only tool in this area, was developed to automatically inject bugs into Ethereum smart contracts. However, *SolidiFI* has the following limitations: (1) although it can be extended to support the injection of other types of bugs, now it can only inject 7 types; (2) its injection accuracy and authenticity are low; (3) The bugs it injects are easier to be detected by detection tools. To address these limitations, we propose an approach called *HuangGai* to enable automatic bug injection for Ethereum smart contracts via analyzing the contracts' control and data flows. Based on this approach, we develop an open-source tool, which can inject 20 types of bugs into smart contracts. Besides, *HuangGai* can determine the locations and styles of injected bugs according to the contract's specific situation, so as to ensure that each injected bug is different. The extensive experiments show that *HuangGai* outperforms *SolidiFI* on the number of injected bug types, injection accuracy, and injection authenticity. The experimental results also reveal that existing analysis tools can only partially detect the bugs injected by *HuangGai*.

Index Terms—Ethereum, Solidity, Smart contract security, Bug injection.

1 Introduction

Smart contracts are autonomous programs running on blockchain [1]. Ethereum is currently the largest platform that supports smart contracts [2]. Lots of applications based on smart contracts have been developed and deployed on Ethereum. Similar to traditional computer programs, it is difficult to avoid bugs in contracts. Recent years have witnessed many attacks that exploit the bugs in smart contracts to cause severe financial loss [3]. Even worse, the deployed contracts cannot be modified for bug patching. Hence, it is essential to detect and fix all bugs in the contracts before deployment.

Recent studies have developed many tools for detecting bugs in smart contract [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]. However, it is difficult to conduct a thorough evaluation of these tools due to the lack of large-scale datasets of smart contracts with diverse bugs. Note that existing evaluations mainly rely on *handwritten contracts* (i.e., contracts that are manually constructed by researchers according to the characteristics of known bugs) [21], [22], [23], [24], [25]. Unfortunately, such buggy smart contracts have the following problems:

- *Lack of real business logic*. Since these manually constructed contracts are only used for assessing the performance of bug detection, they are usually different from *real contracts* in terms of the statement types, control structure types, and programming patterns, etc. For instance, libraries are widely used to create *real contracts*, whereas they are rarely used in *handwritten contracts*. Consequently, such *handwritten contracts* cannot be utilized to truly verify the performance of analysis tools on detecting bugs in *real contracts*.
- *The code size of contracts is generally small*. By investigating 5 widely used datasets with *handwritten contracts* (i.e., [21], [22], [23], [24], [25]) and a dataset with 66,208 *real contracts* collected by us, we find that the average number of *loc* (line of code) per *handwritten contract* is 42, in comparison to 432 *loc* per *real contract*. Such small contracts may lead to biased results when they are used to assess the effectiveness and efficiency of analysis tools for processing *real contracts*.
- *The number of contracts with diverse bugs is inadequate*. The datasets with *handwritten contracts* usually have a small number of samples (around 100 contracts). When it comes to each type of bugs, the contracts that contain certain bugs are much sparser. Hence, the experimental results upon such a small number of contracts with insufficient and unbalanced types of bugs would be biased.

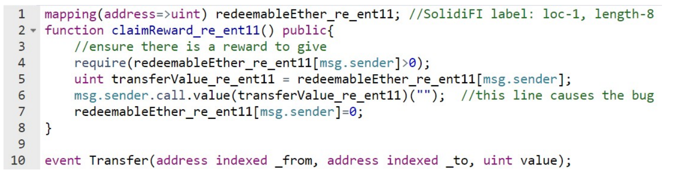
The above problems greatly challenge users to find out the true performance of analysis tools on *real contract* bug detection. An effective way to address the above problems is to inject bugs into the *real contracts* and automatically label the locations and types of the injected bugs. Researchers

- *P. Zhang, F. Xiao are with College of Computer and Information, Hohai University, Nanjing, China.*
E-mail: pchzhang@hhu.edu.cn; harleyxiao@foxmail.com
- *X. Luo is with Department of Computing, the Hong Kong Polytechnic E-mail: csxluo@comp.polyu.edu.hk*
- *H. Dong is with School of Science, RMIT University Melbourne, Australia E-mail: hai.dong@rmit.edu.au*

Manuscript received XXX, XXXX; revised XXX, XXXX.

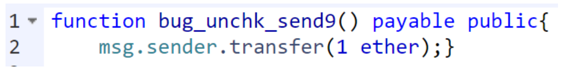
have made efforts in this area [26], [27], among which Ghaleb et al. [27] proposed the first and only Ethereum smart contract bug injection tool, *SolidiFI*. It uses three ways (i.e., *insert full buggy code snippets*, *code transformation*, *weakening security mechanisms*) to inject bugs into all possible positions in a contract. However, *SolidiFI* suffers from the following limitations:

- *Limitations of the SolidiFI approach itself.* By investigating *SolidiFI*, we found that the *SolidiFI* approach itself has the following four limitations:
    1) *The bugs injected by SolidiFI lacks authenticity.* *SolidiFI* claims that it uses three ways to inject bugs into the contracts. However, according to our investigation, most of the bugs are injected by inserting pre-made full buggy code snippets into the contracts, and without taking into account the contracts' original structures (we randomly select 323 *real contracts* and used *SolidiFI* to inject 10,627 bugs into these contracts, and then we found that all bugs were injected by inserting pre-made full buggy code snippets).
    2) *The bugs injected by SolidiFI are easier to be detected by detection tools.* the code snippets inserted by *SolidiFI* are usually simple functions that are independent of the contracts' original control and data flows (e.g., the code snippet shown in Fig 2). Hence, such injected bugs can be easily detected without the need of conducting sophisticated analysis on the contracts.
    3) *SolidiFI can only inject a limited number of bugs.* Most of the bugs injected by *SolidiFI* are duplicates of pre-made code snippets. e.g., although *SolidiFI* claims that it has injected 1,737 *integer overflow and underflow* bugs into 50 contracts, in fact these 1,737 bugs are just duplicates of 40 pre-made code snippets.
    4) *SolidiFI may fail to accurately inject bugs.* Certain bugs injected by *SolidiFI* cannot be exploited by any attacker. For instance, in a code snippet inserted by *SolidiFI* like Fig 1, *SolidiFI* does not insert any statement into the contract to modify the value of the variable *redeemableEther_re_ent11* (declared in line 1). This makes *redeemableEther_re_ent11[msg.sender]* keep the initial value (0) unchanged and the *require-statement* (line 4) always throw an exception. It eventually invalidates the injected bug (line 6).
- *Limitations of SolidiFI's current software engineering.* We found that *SolidiFI* has the following one limitation in software engineering:
    1) *Although SolidiFI can be extended to support the injection of other types of bugs, now it can only inject 7 types.* Existing studies such as [28] show that there are far more than 7 types of bugs in Ethereum smart contracts. If users want to inject other types of bugs into the contracts through *SolidiFI*, they need to prepare the full buggy code snippets by themselves.
- *Limitations of SolidiFI in terms of ease of use.* We found that *SolidiFI* has the following two limitations in terms of ease of use:
    1) The purpose of designing a bug injection tool is to create a large-scale buggy contract data set. However, *SolidiFI* does not provide such data sets, which makes this work does not provide a public unified benchmark for assessing the performance of detection tools.
    2) The complete process of creating buggy contracts is: collect *real contracts*, inject bugs into *real contracts*, and label the types and locations of bugs. However, *SolidiFI* (the tool itself) cannot collect contracts, so users need to prepare the *real contracts* in advance.



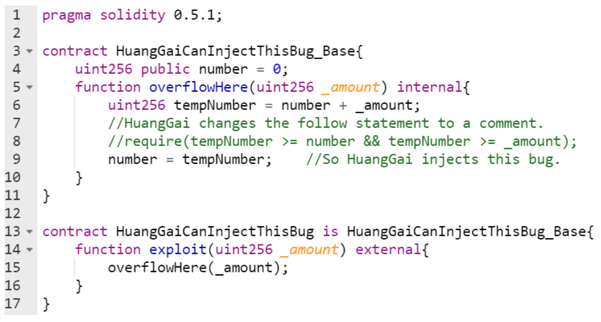
```
1  mapping(address=>uint) redeemableEther_re_ent11; //SolidiFI label: loc-1, length-8
2  function claimReward_re_ent11() public{
3      //ensure there is a reward to give
4      require(redeemableEther_re_ent11[msg.sender]>0);
5      uint transferValue_re_ent11 = redeemableEther_re_ent11[msg.sender];
6      msg.sender.call.value(transferValue_re_ent11)("");  //this line causes the bug
7      redeemableEther_re_ent11[msg.sender]=0;
8  }
9
10 event Transfer(address indexed _from, address indexed _to, uint value);
```

Fig. 1: An example of incapability of *SolidiFI* to accurately inject and precisely label bugs



```
function bug_unchk_send9() payable public{
    msg.sender.transfer(1 ether);}
```

Fig. 2: A code snippet inserted by *SolidiFI* that contains a *wasteful contracts* bug



```
pragma solidity 0.5.1;

contract HuangGaiCanInjectThisBug_Base{
    uint256 public number = 0;
    function overflowHere(uint256 _amount) internal{
        uint256 tempNumber = number + _amount;
        //HuangGai changes the follow statement to a comment.
        //require(tempNumber >= number && tempNumber >= _amount);
        number = tempNumber;    //So HuangGai injects this bug.
    }
}

contract HuangGaiCanInjectThisBug is HuangGaiCanInjectThisBug_Base{
    function exploit(uint256 _amount) external{
        overflowHere(_amount);
    }
}
```

Fig. 3: An *integer overflow and underflow* bug injected by *HuangGai*

To address the above limitations, we propose *HuangGai*[1], a bug injection approach for Ethereum smart contracts. *HuangGai* first collects *real contracts* and extracts control and data flows from the *real contracts*. Then, it checks whether the contracts are suitable for bug injection by analyzing their control and data flows. If that is the case, *HuangGai* identifies the proper locations for bug injection in these contracts and constructs the statements used to inject different types of

1. https://github.com/xf97/HuangGai

bugs. Eventually, up to 20 types of bugs can be injected into the contracts by *HuangGai*.

Our primary contribution is fourfold:

1) We propose an approach called *HuangGai* to automatically inject bugs into Ethereum smart contracts. Based on this approach, we implement an open-source tool which can inject up to 20 types of bugs into contracts. By analyzing the contract's control and data flows, *HuangGai* can ascertain a contract is suitable for injecting which type of bug (this makes *HuangGai* inject bugs by slightly modifying the contract). Besides, *HuangGai* leverages the original structures of the contracts to construct the statements to be inserted into the contract and identifies the proper buggy statement injection locations (this makes the locations and style of the bugs injected by *HuangGai* vary with the contracts). Hereby, *HuangGai* can inject more logical and hard-to-find bugs into the contracts (e.g., the *integer overflow and underflow* bug shown in Fig 3).
2) We conduct extensive experiments to evaluate *HuangGai*. The experimental results show that *HuangGai* can effectively inject more types of bugs with higher accuracy and authenticity, compared to *SolidiFI*.
3) We evaluate a group of state-of-the-art analysis tools [18], [19], [20], [29], [30], [31], [32], [33], [34] by using the buggy contracts generated by *HuangGai*. The detection results show that these analysis tools cannot effectively detect most of the bugs injected by *HuangGai*.Hence, *HuangGai* can be used to uncover the weaknesses in analysis tools.
4) By means of *HuangGai*, we generate and release the following 3 public datasets[2]:
    - *Dataset 1*: This dataset consists of 964 buggy contracts covering 20 types of bugs. These buggy contracts have been thoroughly examined by three contract debugging experts. To the best of our knowledge, *dataset 1* is the largest *real contract* based buggy contract dataset in terms of the number of verified contracts.
    - *Dataset 2*: This dataset comprises 4,744 buggy contracts covering 20 types of bugs, which have not been manually verified. Although the contracts in dataset 2 have not been manually verified, according to our experimental results, more than 97% of the bugs injected by *HuangGai* are accurate. Researchers may employ *datasets 1* and *2* as the benchmark to assess the performance of analysis tools for bug detection.
    - *Dataset 3*: This dataset contains 66,208 *real contracts* without injected bugs. Users can use the contracts in *dataset 3* as raw materials for bug injection.

2. Users can access these three datasets by visiting https://github.com/xf97/HuangGai

The rest of this paper is organized as follows: Section 2 introduces the background. Section 3 presents our approach, *HuangGai*. Section 4 describes the experimental results of *HuangGai*. We outline potential threats to the validity of HuangGai in Section 5. After introducing the related work in Section 6, we conclude the paper with future work in Section 7.

2 BACKGROUND

In this section, the basic concepts used in the paper are introduced. First, the concepts of Ethereum smart contract and Solidity is are described in Section 2.1. Then, smart contract bug detection criteria is introduced in Section 2.2.

2.1 Ethereum smart contract and Solidity

Users deploy smart contracts by sending contract bytecode to Ethereum through transactions. In Ethereum, each contract or user is assigned a unique address as their identifiers. Ether is the default cryptocurrency of Ethereum. Both contracts and users can trade Ethers. Solidity [35] is the most widely used programming language for developing Ethereum smart contract. Users can employ Solidity to develop contracts. A compiler is then utilized to generate the contracts' bytecode. Solidity is a fast-evolving language. New versions of Solidity with breaking changes are released every few months. When developing a contract, developers need to specify the employed Solidity version, so as to use the proper compiler to compile the contract. Solidity assigns a *function selector* to each function. The *function selector* is the first four bytes of the *keccak-256* hash of the signature of the function, it is used to identify the function to be called. It is worth noting that in a few cases, different functions will be assigned the same *function selector* (i.e., hash collision). The overridden function has the same *function selector* value as the overriding function. Solidity supports (multiple) inheritance. Its official compiler (*solc*) can specify linear inheritance orders from base contracts to derived contracts. Solidity provides *require-statement* and *assert-statement* to handle errors. When the parameters of these two types of statements are *false*, *require-statement* or *assert-statement* will throw an exception and rollback the results of program execution.

2.2 Smart contract bug detection criteria

A number of smart contract bug classification frameworks and corresponding detection criteria have been proposed recently [14], [28], [36], [37]. Among them, our previous work [28] proposes a comprehensive smart contract bug classification framework by extending the *IEEE Standard Classification for Software Anomalies* [38], which summarizes 49 types of bugs and their severity levels. Besides, this work [28] also proposes a set of bug detection criteria, i.e., a specific type of bug can be found in a contract as long as the contract matches certain characteristics. In this paper, we focus on how to inject the 20 most severe bug types, such as *re-entrancy* and *integer overflow and underflow*. Table 3 shows the name of each bug type. The specific description and bug detection criteria of each bug type can be found in [28].

3 *HuangGai*

3.1 Overview of HuangGai

HuangGai is an automatic bug injection tool for Ethereum smart contracts. It can inject up to 20 types of severe bugs into the source code of *real contracts*. Specifically, *HuangGai* employs the characteristic criteria proposed in our previous work [28] to create such types of bugs. The constructed buggy smart contracts can be used to evaluate both source code or bytecode based analysis tools.

The workflow of *HuangGai* is shown in Fig 4. *HuangGai* first collects *real contracts* (performed by *ContractSpider* introduced in Section 3.2). It then checks whether a contract is suitable for injecting a certain type of bugs by analyzing the contract's control and data flows, and extracts the data required for injecting bugs (achieved by *ContractExtractor* presented in Section 3.4). According to the data extracted by *ContractExtractor* required for bug injection, *HuangGai* injects bugs into the contract (implemented by *BugInjector* described in Section 3.5).

3.2 ContractSpider

The first step of *HuangGai* is to collect *real contracts* as the sources for bug injection so that users do not need to manually collect the contracts beforehand (of course, *HuangGai* also supports injecting bugs into local contracts). We implement *ContractSpider* based on [39], which is a parallel high-performance web crawler. *ContractSpider* automatically collects contract source code by crawling open-source *real contracts* websites (e.g., http://etherscan.io/), and then saves these source code as Solidity files. Note that *ContractSpider* runs independently of *ContractExtractor* and *BugInjector*. Users can run *ContractSpider* once to collect thousands of *real contracts*, and then users can generate the required buggy contracts by running *ContractExtractor* and *BugInjector* according to their needs.

3.3 Construct a contract's control and data flows

HuangGai analyzes the control and data flows of a contract to check whether the contract is suitable for injecting certain types of bugs. Therefore, we first introduce how *HuangGai* constructs the control and data flows of the contract. In the process of injecting the 20 types of bugs, *HuangGai* uses the same method to construct the contracts' control and data flows.

Based on *solc* [40] and *Slither* [30], *HuangGai* constructs a contract's control and data flows. Specifically, *HuangGai* extracts all the function-call paths from a (derived) contract to track the definitions and use of data in each path. A function-call path refers to a sequence of nodes, each of which denotes a function. Adjacent nodes means that there is a one-way calling relationship between these functions (i.e., the former function calls the latter one).

- **Data flow**: Using *solc* to compile a contract can generate the abstract syntax tree (*AST*) of the contract. By analyzing *AST*, *HuangGai* obtains the following information: *definition-use* pairs [41] of data, and linear inheritance orders. Based on the above information, *HuangGai* tracks the definition and use of all the data.

Algorithm 1 Contract CFG construction algorithm

```
Input: Contract's source code $SC$
Output: function-call paths set $P$
1  changeSolcVersionBySolc($SC$) // Adjust solc version.
2  ContractAstSet $C$ = getContractASTBySolc($SC$) Set $P$
       // function-call paths set.
3  foreach contractAst in $C$ do
       // from base contract to derived contract
4      $callGraphSet = getFuncCallGraphBySlither(SC)$ $CFGSet = getFuncCFGBySlither(SC)$ $contractPathSet = getContractCFG(callGraphSet, CFGSet)$
       $funcAndItsSelector = getFuncSelectorBySolc(contractAst)$
       $P = P \cup contractPathSet$ // Add new function-call paths.
5      foreach path in $P$ do
6          foreach (oldFunc, oldSelector) in path do
7              foreach (newFunc, newSelector) in funcAndItsSelector do
8                  if newSelector == oldSelector then
9                      if newFuncHeader == oldFuncHeader then
                           /* Override */
10                         $oldFunc$ is replaced by $newFunc$
11                     end
12                 end
13             end
14         end
15     end
16 end
17 return $P$
```

- **Control flow**: *HuangGai* generates a (derived) contract's *CFG* by using the control flow graph (*CFG*) of each function and the function-call graph generated by *Slither* as well as the linear inheritance order generated by *solc*. Algorithm 1 shows this process. When a function (or function modifier) overrides the same function (or function modifier) of the base contract, *HuangGai* uses the following two steps to identify the overridden function on the path and replaces it with the overriding function: (1) If the *function selectors* of these two functions are different, *HuangGai* will consider that there is no function override. If not, *HuangGai* enters step 2. (2) *HuangGai* reads the contract's source code, gets the function headers of these two functions, and compares whether the headers are the same. If these two headers are same, *HuangGai* will consider that there is an override of the derived class function to the base class function.

Another detail worthy of explanation is the path reachability problem. After constructing function-call paths, *HuangGai* should check the reachability of these paths. However, due to the following two reasons, *HuangGai* does not check the reachability of these paths: 1) contracts rarely contain dead code (i.e., unreachable code statements). 2) we need to accelerate the injection speed of *HuangGai*. To reduce the impact of not checking the reachability of these paths, through a survey of hundreds of randomly selected contracts, we found that access control statements (e.g, *require(msg.sender == owner);*) are the main reason for unreachable paths (i.e., the same contract has different path reachability to different users). Therefore, *HuangGai* will invalidate the access control statements on these paths (by changing the conditional judgment part into the ever-true expression, e.g, *require(true == true);*).

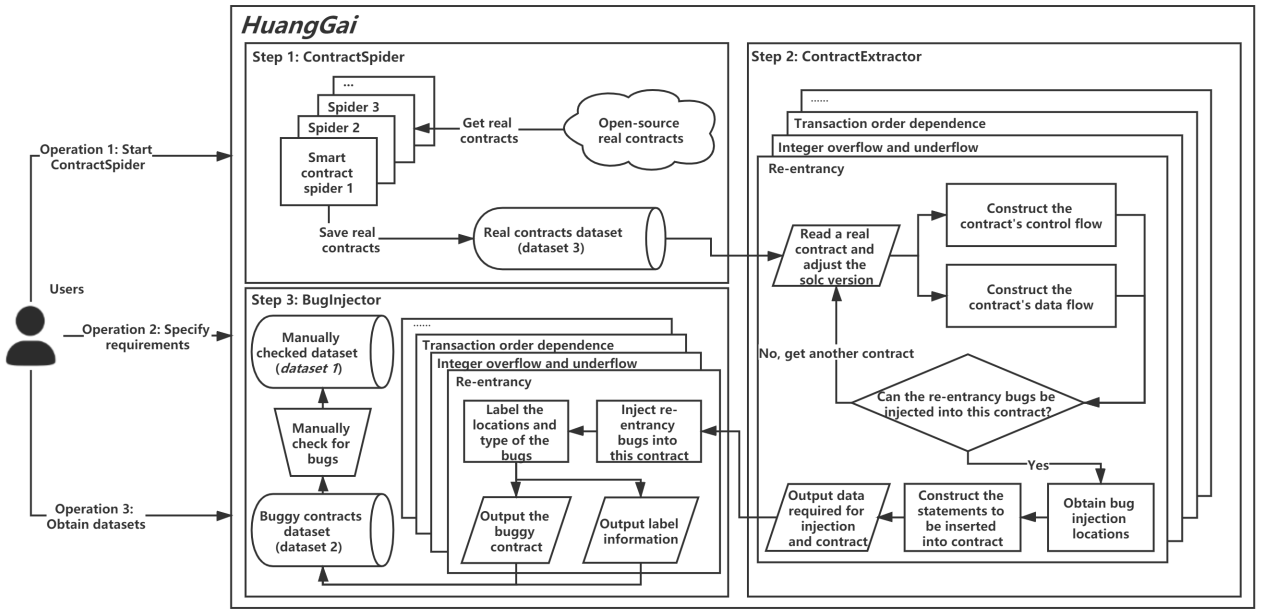

HuangGai
Step 1: ContractSpider
...
Spider 3
Spider 2
Smart contract spider 1
Get real contracts
Open-source real contracts
Save real contracts
Real contracts dataset (dataset 3)
Step 2: ContractExtractor
......
Transaction order dependence
Integer overflow and underflow
Re-entrancy
Read a real contract and adjust the solc version
Construct the contract's control flow
Construct the contract's data flow
No, get another contract
Can the re-entrancy bugs be injected into this contract?
Yes
Obtain bug injection locations
Construct the statements to be inserted into contract
Output data required for injection and contract
Operation 1: Start ContractSpider
Users
Operation 2: Specify requirements
Operation 3: Obtain datasets
Step 3: BugInjector
Manually checked dataset (dataset 1)
Manually check for bugs
Buggy contracts dataset (dataset 2)
......
Transaction order dependence
Integer overflow and underflow
Re-entrancy
Label the locations and type of the bugs
Inject re-entrancy bugs into this contract
Output the buggy contract
Output label information

Fig. 4: *HuangGai* workflow

3.4 ContractExtractor

One problem of injecting bugs by inserting fully buggy code snippets into *real contracts* is that it is difficult to inject a large number of different complex bugs. This is because triggering complex bugs often requires multiple operations (e.g, variable declaration and initialization, variable state changes, and triggering bugs), and it is often difficult for code snippet authors to write a large number of different code snippets containing complex bugs. One way to solve this problem is to only inject a certain type of bug into the contract that is suitable for injecting that type of bug. For instance, the contract (*reentrancy*) shown in Fig 5 is a contract suitable for injecting *re-entrancy* bugs. *HuangGai* only needs to construct a statement (e.g, the statement in line 16) and insert this statement before the statement in line 18 to inject a *re-entrancy* bug. Hence, *HuangGai* first checks if a contract is qualified for being injected with a certain type of bugs.

ContractExtractor is designed to achieve this goal. *ContractExtractor* takes in a contract collected by *ContractSpider*, extracts the contract's control and data flows, and analyzes the control and data flows to check whether a certain type of bugs can be injected into the contract based on the predefined extraction criteria. If a contract is qualified for being injected with a certain type of bugs, *ContractExtractor* will pass the contract and the data required for bug injection to *BugInjector*; otherwise *ContractExtractor* will skip this contract. It is noteworthy that for different types of bugs the internal workflows and the employed techniques of *ContractExtractor* are distinct. In this paper, we use two representative bugs, namely *re-entrancy* and *integer overflow and underflow*, as examples to describe in detail how *ContractExtractor* works, and we briefly introduce *ContractExtractor(s)* of the remaining 18 types of bugs in Table 1.

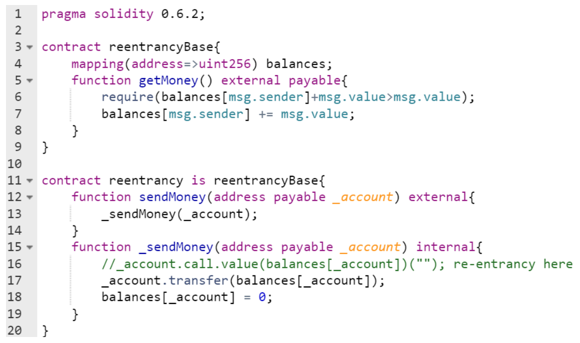

```
1  pragma solidity 0.6.2;
2  
3  contract reentrancyBase{
4      mapping(address=>uint256) balances;
5      function getMoney() external payable{
6          require(balances[msg.sender]+msg.value>msg.value);
7          balances[msg.sender] += msg.value;
8      }
9  }
10 
11 contract reentrancy is reentrancyBase{
12     function sendMoney(address payable _account) external{
13         _sendMoney(_account);
14     }
15     function _sendMoney(address payable _account) internal{
16         //_account.call.value(balances[_account])(""); re-entrancy here
17         _account.transfer(balances[_account]);
18         balances[_account] = 0;
19     }
20 }
```

Fig. 5: A contract that *HuangGai* can inject a *re-entrancy* bug

3.4.1 ContractExtractor for re-entrancy (CER)

According to [28], the characteristic of *re-entrancy* bugs is that the *call-statement* that does not specify a response function is used to send ethers to the payee, and the *call-statement* is executed before the *deduct-statement* (i.e, the statement that deduct the number of tokens held by the payee address). This allows the payee to withdraw several times the same amount of ethers that he(she) deposited into the contract and even drain the contract's ethers. Since *HuangGai* can insert the *call-statements* that do not specify a response function to send ethers to payee , the key to injecting *re-entrancy* bugs is to find the *deduct-statements*. To achieve it, *HuangGai* first locates the variables that record the relationship between the addresses and the number of tokens held by the addresses (we call such variables as *ledgers*).

CER searches for the *ledgers* and *deduct-statements* of a contract through the following steps:

- **Step 1**: *CER* searches for the *deposit paths* in the contract. The *deposit path* refers to a function-call path in the contract meeting the following con-

TABLE 1: The *contractExtractor(s)* and *bugInjector(s)* of the remaining 18 types of bugs

Bug type	**ContractExtractor**	**BugInjector**
Transaction order dependence	The contract (**Con**) needs to be developed based on *ERC-20* token standard and contains the *approve* function	*HuangGai* **(HG)** invalidates security measures to allow the quota of the approved address to be set from one nonzero value to another nonzero value, and labels the assignment statement as a bug
Results of contract execution affected by miners	**HG** searches for the *if-statements* in **Con** that meet the following conditions: the condition part of the *if-statement* is one of the following types: *bytes32*, *address payable*, *uint256* or *address*	**HG** replaces an operand in the condition part of the *if-statement* with the following global variables: *block.coinbase* (address type), *block.coinbase* (address payable type), *block.gaslimit* (uint256 type), *block.number* (uint256 type), *block.timestamp* (uint256 type), *blockhash(block.number)* (bytes32 type), and labels the *if-statement* as a bug
Unhandled exception	**Con** needs to contain at least one of the *low-level call statement* (i.e., *call-statement*, *send-statement*, and *delegatecall-statement*)	**HG** uses the following 2 ways to invalidate the security measures of the low-level call statement: receiving the return value but not checking the return value, or not receiving the return value. **HG** labels the *low-level call statement* as a bug
Use *tx.origin* for authentication	**HG** captures the *address type* variables assigned in the *constructor* function (we call these variables *ownerCandidate*) and then searches for *bool* expressions such as (*ownerCandidate ==(!=) address type variable*) in the contract. **Con** needs to contain at least 1 *bool* expressions that meets the above conditions	**HG** replaces *address type* variable (non-*ownerCandidate*) in the *bool* expression that meets the extraction criteria with *tx.origin*, and labels the *bool* expression as a bug
Wasteful contracts	**Con** needs to contain functions with the following conditions: the function's visibility is *public* or *external*, and the function needs to contain the statements that can transfer out ethers	**HG** invalidates all statements in the above functions (including the function modifiers) that may interrupt the execution of these functions, and inserts a statement to transfer out all ethers of **Con** at the end of the function (e.g., *msg.sender.transfer(address (this.balance);)*). Finally **HG** labels the statement that can transfer out ethers as a bug
Short address attack	**Con** needs to contain functions that meet the following conditions: 1) the visibility is *external* or *public*, 2) the function needs to contain statements that can transfer out ethers, and 3) the payee address and the number of ethers transferred are provided by the function caller	**HG** invalidates all check statements (i.e., statements that check *msg.data.length*) in the above functions (and the function modifiers), and labels these functions as bugs
Suicide contracts	**Con** needs to contain functions that meet the following conditions: visibility is *external* or *public*, and functions need to contain *self-destruct-statements* (e.g., *suicide-statement* and *selfdestruct-statement*)	**HG** invalidates all statements in the above functions (and the function modifiers) that may interrupt execution or authenticate, and labels the *self-destruct-statements* as bugs
Locked ether	**Con** needs to contain functions declared as *payable*	**HG** invalidates all the statements used to transfer out ethers in **Con** in the following 2 ways: setting the number of transferred out to 0, or changing the *transfer-out-statements* (i.e, *send-statement*, *transfer-statement*, *call-value-statement*) to comments, and label the contract as containing this type of bug
Forced to receive ether	**Con** needs to contain at least 1 of the 3 types of statements: *if-statement*, *require-statement*, or *assert-statement* that meet the following conditions: the type of the condition part of the statement is *uint256*, and at least one operand in the condition part is *uint256* literals	**HG** will replace the *non-uint256-literals* operand in the condition part of the above statements with *address(this).balance*, and label these statements as bugs
Pre-sent ether. According to [28], *Forced to receive ether* bug and *Pre-sent ether* bug are the manifestations of the same type of bugs in different periods. Therefore, the *ContractExtractor(s)* of these 2 types of bugs are the same. The difference is that the bug types labeled by the *BugInjector(s)* of the 2 types of bugs are different		
Uninitialized local/state variables	**Con** needs to contain initialization statements for local variables or state variables (non-constant-varibales)	**HG** invalidates the assignment part in the initialization statement (e.g., *uint num; //= 1;*), and labels the initialization statement as a bug
Hash collisions with multiple variable length arguments	**Con** needs to contain functions that meet the following conditions: 1) visibility is *external* or *public*, 2) functions contain multiple array type parameters, and 3) function contains the declaration statement for the *bytes32* variable	**HG** re-assigns the *bytes32* variables in the above functions to *keccak256(abi.encodePacked(para))*, where the *para* are the array parameters passed by the external callers, and then labels the *bytes32* assignment statement as a bug
Specify *function* variable as any type	**Con** needs to contain *function* type variables	**HG** inserts *assembly-statements* so that external attackers can modify the type of *function* variables and then labels the inserted *assembly-statements* as bugs
Dos by complex *fallback* function	**Con** needs to contain the *fallback* functions that meet the following condition: the *fallback* function contains statements that can transfer out ethers	**HG** inserts the statements at the end of the above *fallback* functions (e.g., *payee address.call.gas(2301).value(1)("");*), where the *payee* address is the payee address in *fallback* function. Then **HG** labels the *fallback* functions as bugs
Public function that could be declared *external*	**Con** needs to contain functions with *external* visibility	**HG** changes the visibility of the above functions from *external* to *public* and labels these functions as bugs
Non-public variables are accessed by *public/external* function	**Con** needs to contain *external/public* functions	**HG** searches for whether the above functions contain access operations to *private/internal* state variables with visibility. If so, **HG** will label these access operations as bugs
Nonstandard naming	**Con** needs to contain at least 1 of the following 4 types of structures: *function*, *function modifier*, *event* and *constant variable*	**HG** changes the naming of these structures to non-standard form, and then modifies the naming of these structures in **Con**. Then **HG** labels the declaration statements of structures as bugs
Unlimited compiler versions	**Con** needs to contain the*pragma-solidity-statement*	**HG** analyzes the *pragma-solidity-statements* to get the oldest Solidity version that can compile the contract, and then replaces the original *pragma-solidity-statements* with the following statement: *pragma solidity ∧ minimum Solidity version*. And **HG** labels the new *pragma-solidity-statements* as bugs

ditions: 1) The entry function of this path is a *public* function declared as ***payable***. 2) There is at least one value increment operation on a *mapping(address=>uint256)* variable in this path. We call the *mapping(address=>uint256)* variable as a *potential ledger*. We call the set of all the *potential ledgers* in the contract as a *potentialLedgerSet*.

- **Step 2**: *CER* searches for *withdrawal paths* in the contract. The *withdrawal path* refers to a function-call path in the contract meeting the following conditions: 1) There is at least one value decrement operation on a *potential ledger* in this path. We call a *potential ledger* that meets this condition as a *target ledger*. 2) There is at least one operation to send ethers in this path, where the payee address needs to be same as the address of the value decrement operation in the *target ledger*. We call the set of all the *target ledgers* in the contract as a *ledgerSet*, and the set of locations for all the value decrement operations in the *withdrawal path* as a *deductSet*.
- **Step 3**: All variables in the *ledgerSet* can be regarded as the *ledgers* in the contract and the *deductSet* that records the locations of all *deduct-statements*.

If a contract's *deductSet* is not empty, *HuangGai* only needs to insert the *call-statements* for sending ethers in front of all the locations in the *deductSet* to inject *re-entrancy* bugs into the contract. For instance, in the contract shown in Fig 5, there exist both a *deposit path* (func **getMoney**) and a *withdrawal path* (func **sendMoney**, func **_sendMoney**). The *ledgerSet* of this contract is $\{balances\}$, and the *deductSet* of this contract is {line 17}. *HuangGai* only needs to insert a *call-statement* for sending ethers in line 16 to inject a *re-entrancy* bug.

By analyzing the control and data flows of a contract, *HuangGai* can construct the contract's *potentialLedgerSet*, *ledgerSet*, and *deductSet*. If a contract's *deductSet* is not empty, *CER* will pass the source code, payee addresses, and *deductSet* of the contract to the *BugInjector* of *re-entrancy* for conducting bug injection.

3.4.2 ContractExtractor for integer overflow and underflow (CEI)

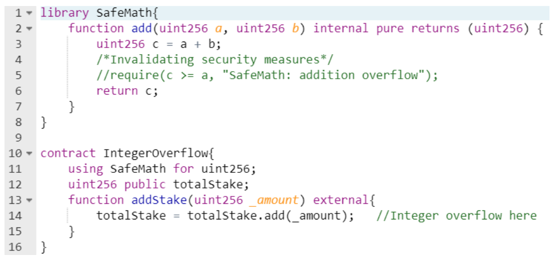
According to [28], the characteristics of the *integer overflow and underflow* (*IOA*) bug are:

- *Characteristic 1*: The maximum (or minimum) values that are generated by the operands participating in the integer arithmetic statement can exceed the storage range of the result.
- *Characteristic 2*: The contract does not check whether the result is overflow or underflow.

To construct the above characteristics, *CEI* needs to find integer arithmetic statements fulfilling the following conditions in a contract:

- *Condition 1*: The types of the operand variables and the result variable are same. This condition can ensure that the statement meets *characteristic 1*.
- *Condition 2*: In the integer arithmetic statement, at least one operand variable is a parameter passed by the external caller. This condition can ensure that the injected bugs can be exploited by external attackers.

We call the integer arithmetic statements that meet *condition 1* and *condition 2* as *target statements*. As shown in Fig 6, when a *target statement* is found (line 14), *HuangGai* only needs to invalidate the corresponding *check statement* (line 5, i.e., the statement used to check whether the result is overflow or underflow) to inject an *IOA* bug. *CEI* identifies the *check statements* based on our following experience. Generally speaking, *require-statements* or *assert-statements* are used to check the results of integer arithmetic statements. Therefore, if the operands participating in the integer arithmetic statements are the parameters of a *require-statement* or an *assert-statement*, *CEI* regards this *require-statement* (or *assert-statement*) as a *check statement*.



```
1  library SafeMath{
2      function add(uint256 a, uint256 b) internal pure returns (uint256) {
3          uint256 c = a + b;
4          /*Invalidating security measures*/
5          //require(c >= a, "SafeMath: addition overflow");
6          return c;
7      }
8  }
9
10 contract IntegerOverflow{
11     using SafeMath for uint256;
12     uint256 public totalStake;
13     function addStake(uint256 _amount) external{
14         totalStake = totalStake.add(_amount);   //Integer overflow here
15     }
16 }
```

Fig. 6: How *HuangGai* invalidates security measures.

CEI searches for the *target statements* and *check statements* in the contract according to the following steps:

- **Step 1**: *CEI* searches for integer arithmetic statements or statements that use library functions for integer arithmetic. We call the set of search results as a *candidate*.
- **Step 2**: *CEI* verifies whether each statement in the *candidate* meets *condition 1* and *condition 2*. We call the set of statements in the *candidate* that meets the two conditions as a *target*.
- **Step 3**: *CEI* checks each statement in the *target* and finds the corresponding *check statement* by analyzing the contract's control and data flows, i.e., determining whether a *check statement* is in the same function or the library function. We call the set of *check statements* as an *opponent*.

When the *target* of a contract is not empty, *HuangGai* can inject *IOA* bugs into the contract and pass the source code, *target*, and *opponent* of the contract to the *BugInjector* of *IOA*.

3.5 BugInjector

A *BugInjector* receives a contract and the data required for injecting a specific type of bugs from its corresponding *ContractExtractor* and performs bug injection and labeling. *BugInjector* always modifies the content of the contract. One of our goal is to keep the authenticity of the *injected contract* (i.e., contracts with injected bugs) as much as possible. First, we define the *real contract*: a contract written by developers, not for the purpose of containing a certain bug (but for achieving a certain commercial or non-commercial purpose), and deployed on Ethereum is a *real contract*. According to our definition, any modification of the *real* *contract* by the bug injection tool will affect the authenticity. So, the less the bug injection tool modifies the contract content, the more the *injected contract* keeps its authenticity. Hence, to keep the authenticity of the *injected contract* as much as possible, *BugInjector* only will slightly modifies the contract.

According to the received data, the *BugInjector* injects and labels a type of bugs by one of the following means:

- Inserting statements that cause bugs. *BugInjector* inserts the statements that cause the specific type of bugs into the contract and labels the bugs in the insertion locations. *HuangGai* will use the structures of the contract that are visible in the current scope to construct the inserted statements, so as to reduce the probability of compilation errors and keep the style of the inserted statements consistent with the original code of the contract.
- Invalidating security measures. *BugInjector* first invalidates the security measures in the contract, followed by labeling the statements without security protection as having bugs. Generally speaking, the security measures invalidated by *HuangGai* are error-handling statements (e.g, *require-statement* and *assert-statement*), such statements are usually used to check the contract state and roll back the transaction when the contract state is abnormal.

In this paper, we use the *BugInjectors* of *re-entrancy* and *integer overflow and underflow* as examples to describe in detail how *BugInjector* works, and we briefly describe the *BugInjector*(s) of the remaining 18 types of bugs in Table 1. These two *BugInjectors* use the aforementioned means to inject bugs.

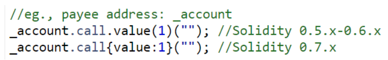

```
//eg., payee address: _account
_account.call.value(1)(""); //Solidity 0.5.x-0.6.x
_account.call{value:1}(""); //Solidity 0.7.x
```

Fig. 7: How *HuangGai* constructs *call-statements* based on a payee address.

3.5.1 BugInjector of re-entrancy (BIR)

BIR injects *re-entrancy* bugs by inserting statements that cause bugs. *CER* passes the following information to *BIR*: contract source code, payee addresses, and locations of *deduct-statements* (*deductSet*). *BIR* uses the payee addresses to construct the *call-statements* for sending ethers (as shown in Fig 7) and inserts the *call-statements* in front of the *deduct-statements* based on the *deductSet*. Finally, it labels the lines where the *call-statements* are inserted as having *re-entrancy* bugs. For instance, in the contract shown in Fig 5, *HuangGai* will insert a *call-statement* (i.e., *_account.call.value(1)("")*) in front of the code in line 18, and label the inserted line as having a *re-entrancy* bug.

3.5.2 BugInjector of integer overflow and underflow (BII)

BII injects *IOA* bugs by invalidating security measures. *CEI* passes the following information to *BII*: contract source code, *target*, and *opponent*. *target* contains the locations of the statements that may cause *IOA* bugs. *opponent* contains the locations of *check statements*. *BII* invalidates all the *check statements* (by changing the statements to comments) in the *opponent* to make the statements in the *target* without security protection. Next, *BII* labels the statements in the *target* as having *IOA* bugs. For instance, in the contract shown in Fig 6, *HuangGai* will change the code in line 5 to a comment (assuming the code in line 5 is not commented), and then label the code in line 14 as having an *IOA* bug.

4 Evaluation

Comparing *HuangGai* with state-of-the-art tools, we conduct extensive experiments to answer four research questions:

- **RQ1**: Can *HuangGai* inject bugs more accurately?
- **RQ2**: Can *HuangGai* inject more authentic bugs?
- **RQ3**: Can users find more weaknesses in existing analysis tools by using *HuangGai*?
- **RQ4**: Can *HuangGai* inject bugs more efficiently?

4.1 Tools used in the evaluation

We employ two types of tools: 1) State-of-the-art Ethereum smart contract bug injection tools. To the best of our knowledge, there is currently only *SolidiFI* available in this area. We compare the performance of *HuangGai* and *SolidiFI*. 2) State-of-the-art smart contract analysis tools. We apply these analysis tools to detect the buggy smart contracts generated by *HuangGai* and *SolidiFI*.

TABLE 2: Selected and excluded tools based on our selection criteria

	Selection criteria	Tools that violate the criteria
Excluded	criterion 1	*teEther,Zeus,ReGuard, SASC, Remix, sCompile, Ether, Gasper, sFuzz*
	criterion 2	*Vandal, Echidna, MadMax, VeriSol, ILF*
	criterion 3	*EthIR, E-EVM, Erays, Ethersplay, EtherTrust, contractLarva, FSolidM, KEVM, SolMet, Solhint, rattle, Solgraph, Octopus, Porosity, Verx*
	criterion 4	*Osiris, Oyente, HoneyBadger*
Selected	*Maian, Manticore, Mythril, Securify, Slither, SmartCheck*	

The state-of-the-art smart contract analysis tools are collected via two channels: 1) Analysis tools that have been covered by the latest empirical review papers, i.e., [21], [27]. 2) Analysis tools that are available on GitHub [42]. We use the keywords *smart contract security* and *smart contract analysis tools* to search in Github and select the twenty tools with the highest numbers of *stars* from the search results.

Not all analysis tools are applicable for our evaluation. We select analysis tools from the collected results based on the following criteria:

- *Criterion 1*. It supports command-line interface so that we can apply it to buggy contracts automatically.
- *Criterion 2*. Its input is Solidity source code or bytecode.
- *Criterion 3*. It is a bug detection tool, i.e., this tool can report types and locations of bugs in contracts.

- *Criterion 4*. It can analyse contracts written in Solidity 0.5.x (i.e., 0.5.0-0.5.16). This is because *SolidiFI* can inject bugs into contracts written in Solidity 0.5.0 and older versions, but *HuangGai* currently supports injecting bugs into contracts written in Solidity 0.5.0 and subsequent versions. So, to ensure fairness, we will use the contracts that are written in Solidity 0.5.x for our evaluation.

According to the selection criteria, we finally select 6 analysis tools (*Maian* [33], *Manticore* [43], *Mythril* [31], *Securify* [34], *Slither* [30], *SmartCheck* [29]) to detect bugs injected by *HuangGai* and *SolidiFI*. Table 2 lists the 6 selected tools.

4.2 Dataset for evaluation and environment

We first employ *HuangGai* and *SolidiFI* to generate an *injected contract* dataset. The original *real contracts* are sourced from the Solidity 0.5.x contracts in *dataset 3* described in the introduction. We then evaluate the bug injection performance of the two tools by inspecting each contract in the *injected contract* datasets. To ensure fairness, we use *HuangGai* and *SolidiFI* to inject the same type of bugs into the same contracts to generate the dataset. It is worth noting that, unlike *SolidiFI*, *HuangGai* cannot inject any type of bugs into each contract. Specifically, for each type of bugs, we first use *HuangGai* to generate 50 *injected contracts*, and then utilize *SolidiFI* to inject the same bugs into the same group of contracts. Finally, we obtain a dataset comprising 897 *injected contracts* (covering 20 types of bugs) generated by *HuangGai* and 322 *injected contracts* (covering 7 types of bugs) generated by *SolidiFI*. This is because *HuangGai* cannot find 50 qualified contracts from the dataset 3 for certain types of bugs. Besides the above two datasets, we also generate unlabeled versions of them (i.e., *injected contracts* without bug labels). The reason we generate the unlabeled versions is: It is not trivial to evaluate the performance of bug injection tools. If we only use the bug detection tools to evaluate (e.g, the more bugs that are detected by detection tools, the more accurate the bugs generated by an injection tool). Assuming that all bugs generated by a bug injection tool will be detected by the detection tools, it is meaningless to design and develop this injection tool (i.e., this injection tool cannot reveal the weaknesses of the detection tools). Therefore, we will use a combination of human experts and detection tools to evaluate the performance of injection tools. However, both *HuangGai* and *SolidiFI* will insert labels (in the form of the single-line comments) into the *injected contract* to indicate the locations and types of the injected bugs, this will make it easy for human experts to find the injected bugs. So, we need to generate the unlabeled version.

Our evaluation environment is built upon a desktop computer with Ubuntu (18.04) operating system, AMD Ryzen5 2600x CPU, 16GB memory, and NVIDIA GTX 1650 GPU.

4.3 RQ1: Bug injection accuracy

We calculate the accuracy of bug injection as follows:

$$accuracyRate = (BIN - IABN) \div BIN \quad (1)$$

where *IABN* represents the number of bugs that cannot be activated, and *BIN* represents the number of bugs injected by the bug injection tool. We calculate the value of *IABN* by checking the following two aspects:

- whether the *injected contract* can be compiled. Since the bug injection tool always modifies the content of the contract, we use *solc* of the original contract to compile the *injected contract*. If there are compilation errors in an *injected contract*, all the injected bugs in the contract are not deemed to be activated.
- whether the injected bugs can be exploited by attackers and cause the expected consequences. Three smart contract debugging experts manually check each injected bug on the premise of knowing the locations and types of injected bugs, and reach a consensus through discussion. If a bug cannot be exploited, it cannot be activated.

Result. In this experiment, we use the original version of the dataset (i.e., *injected contracts* with bug labels). Table 3 shows the *accuracyRate*(s) of *HuangGai* and *SolidiFI* for bug injection. *N/A* means that the bug injection tool is not designed to inject a type of bugs. We can see that *HuangGai* can inject more types of bugs with higher accuracy, compared with *SolidiFI*. Specifically, *HuangGai* shows **equal or higher** *accuracyRate(s)* than *SolidiFI* for all the 7 comparable types of bugs.

Analysis. Through asking debugging experts, we summarize and list the reasons for the injection failure of *SolidiFI* and *HuangGai*. The main reason for the injection failures of *SolidiFI* is that the inserted code snippets contain dead code. e.g., Fig 8 shows a fully code snippet inserted by *SolidiFI*, and the purpose of this snippet is to inject a *transaction order dependence* bug into the contract. The original intention of this code snippet author is that the different execution orders of function *setReward_TOD22* and *claimReward_TOD22* will lead to different results (e.g, executing *claimReward_TOD22* before *setReward_TOD22* will cause the statement in line 5 to always throw an exception, while executing *setReward_TOD22* first will not). However, because *SolidiFI* does not assign a suitable value to the variable *owner_TOD22*, which makes *owner_TOD22* keep the default initial value (*0x0*), and causes the *require-statement* in line 7 to always throw an exception. This eventually makes the statements in lines 8 and 9 dead code and makes executing function *setReward_TOD22* meaningless.

The main reasons for the injection failures of *HuangGai* is the excessive security measures used in some contracts. In some contracts, developers use multiple redundant security measures to prevent bugs. Although *HuangGai* can invalidate most of the common security measures (we obtain the common security measures of each type of bug by investigation), some extremely rare security measures prevent the injected bugs from being activated.

4.4 RQ2: Bug injection authenticity

It is difficult to quantitatively evaluate the authenticity of injected bugs. One way to measure the authenticity of injected bugs is to manually inspect the *injected contracts* by experienced smart contract developers and record the bugs they identify. If the injected bugs lack authenticity, developers can easily identify them. Therefore, another

TABLE 3: The *accuracyRate(s)*, *realRate(s)* and *speed(s)* of *HuangGai(HG)* and *SolidiFI(SF)* for bug injection

	accuracyRate		***realRate***		***speed***			***accuracyRate***		***realRate***		***speed***	
Bug type	*HG*	*SF*	*HG*	*SF*	*HG*	*SF*	Bug type	*HG*	*SF*	*HG*	*SF*	*HG*	*SF*
Transaction order dependence	96.0%	75.1%	3.54	2.16	34.1	0.5	Forced to receive ether	100.0%	N/A	2.98	N/A	4.8	N/A
Results of contract execution affected by miners	100.0%	97.9%	3.35	2.07	1.2	1.7	Pre-sent ether	100.0%	N/A	2.96	N/A	5.1	N/A
Unhandled exception	100.0%	97.3%	3.17	2.41	3.4	0.6	Uninitialized local/state variables	99.9%	N/A	3.24	N/A	0.7	N/A
Integer overflow and underflow	98.1%	97.7%	3.21	2.35	4.0	1.2	Hash collisions with multiple variable length arguments	97.5%	N/A	3.48	N/A	84.0	N/A
Use *tx.origin* for authentication	100.0%	81.9%	2.87	1.91	4.1	0.5	Specify *function* variable as any type	100.0%	N/A	3.67	N/A	1949.8	N/A
Re-entrancy	96.7%	0.0%	3.01	#	352.1	0.4	Dos by complex *fallback* function	98.0%	N/A	3.16	N/A	61.2	N/A
Wasteful contracts	98.0%	91.7%	3.46	2.37	5.6	0.8	*Public* function that could be declared *external*	100.0%	N/A	3.08	N/A	1.3	N/A
Short address attack	100.0%	N/A	3.67	N/A	17.1	N/A	Non-public variables are accessed by *public/external* function	98.7%	N/A	3.29	N/A	0.9	N/A
Suicide contracts	96.9%	N/A	3.01	N/A	209.8	N/A	Nonstandard naming	99.8%	N/A	2.56	N/A	1.7	N/A
Locked ether	100.0%	N/A	3.24	N/A	3.9	N/A	Unlimited compiler versions	100.0%	N/A	2.45	N/A	7.2	N/A

```
bool claimed_TOD22 = false;
address payable owner_TOD22;
uint256 reward_TOD22;
function setReward_TOD22() public payable {
    require (!claimed_TOD22);

    require(msg.sender == owner_TOD22);
    owner_TOD22.transfer(reward_TOD22);
    reward_TOD22 = msg.value;
}

function claimReward_TOD22(uint256 submission) public {
  require (!claimed_TOD22);
  require(submission < 10);

  msg.sender.transfer(reward_TOD22);
  claimed_TOD22 = true;
}
```

Fig. 8: A *transaction order dependence* bug injected by *SolidiFI* that cannot be activated

three smart contract debugging experts manually inspect the *injected contracts* without knowing the locations and types of injected bugs, and record the bugs they identify. Considering the workload of the manual inspection, we randomly select 30% of the *injected contracts* (these contracts do not contain bugs that cannot be activated) for the experts to inspect. Specifically, we use a scoring method to quantify the authenticity of injected bugs. If these experts do not identify a injected bug, we will give the authenticity of this bug a score of 4; if the experts identify a injected bug, and then the experts will score based on the difficulty of identifying this bug, with a maximum of 3 points and a minimum of 1 point (the higher the score, the more difficult it is to identify the bug).

The following formula is used to calculate the authenticity of the injected bugs:

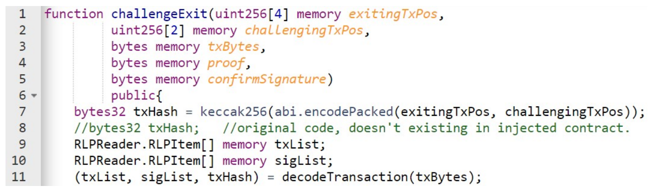
$$realRate = \sum_{n=1}^{BAN} score \div BAN \qquad (2)$$

where *score* represents the score of each injected bug, and *BAN* represents the number of bugs that can be activated injected by the bug injection tool. The higher the *realRate(s)*, the more difficult it is for debugging experts to identify the bugs injected by the bug injection tool, i.e., the higher the authenticity of the injected bugs.

Result. In this experiment, we use the unlabeled version of the dataset. Table 3 shows the *realRate(s)* of *HuangGai* and *SolidiFI* for bug injection. *N/A* denotes that the bug injection tool is not designed to inject a type of bugs. # represents that a type of bugs injected by the bug injection tool cannot be activated. It can be seen from Table 3 that *HuangGai*'s *realRate(s)* are significantly ahead of *SolidiFI*'s *realRate(s)*, which means that compared to *SolidiFI*, the bugs injected by *HuangGai* are more authentic.

Analysis. By asking debugging experts and sampling the *injected contracts*, we summarize and list the reasons for the different authenticity of bugs injected by *SolidiFI* and *HuangGai*. The main reasons why the bugs injected by *SolidiFI* can be easily identified by debugging experts are: 1) *SolidiFI* mainly employs pre-made code snippets without considering the existing structures of the contract to inject bugs, and thus the code style (e.g., naming) of its injected code snippets is quite different from that of the target contract. 2) Each code snippet inserted by *SolidiFI* usually only includes a simple function, which is independent of the contract's original control and data flows. The inserted code snippets are isolated between each other and usually only contain one or more bugs of the same type. 3) Most of the bugs injected by *SolidiFI* are duplicates of pre-made code snippets. e.g., although *SolidiFI* claims that it has injected 1,737 *IOA* bugs into 50 contracts, in fact these 1,737 bugs are just duplicates of 40 pre-made code snippets. These reasons enable debugging experts to easily identify injected bugs without the need of conducting an in-depth analysis of the *injected contract*.

The main reasons why the bugs injected by *HuangGai* are difficult to be identified by debugging experts are: 1) *HuangGai* uses the original structures of the contract (e.g., variables that have been declared and visible in the current scope) to construct the statements to be inserted into the contracts, which makes it difficult for the debugging experts to quickly identify the bugs by skimming the code. 2) *HuangGai* injects a bug into a contract by slightly modifying the contract's content. 3) The bugs injected by *HuangGai* can result from joint actions of two or more functions defined in multiple contracts, which increases the difficulty of bug detection to the next level. These reasons enable debugging experts need to conduct (multiple) in-depth analysis of the *injected contract* to be able to identify the bugs injected by *HuangGai*. e.g., Fig 9 shows a *hash collisions with multiple variable length arguments* bug injected by *HuangGai* (line 7). The statement in line 8 does not actually exist in the *injected contract*, and this statement is the original statement in line 7 before *HuangGai* automatically modifies the contract. It can be seen from Fig 9, *HuangGai* finds a function suitable for injecting *hash collisions with multiple variable length arguments* bugs by analyzing the contract, constructs the statement to be inserted using the variables in the contract, and finally inserts the statement into the proper position to inject a bug.



```
1  function challengeExit(uint256[4] memory exitingTxPos,
2          uint256[2] memory challengingTxPos,
3          bytes memory txBytes,
4          bytes memory proof,
5          bytes memory confirmSignature)
6          public{
7      bytes32 txHash = keccak256(abi.encodePacked(exitingTxPos, challengingTxPos));
8      //bytes32 txHash;   //original code, doesn't existing in injected contract.
9      RLPReader.RLPItem[] memory txList;
10     RLPReader.RLPItem[] memory sigList;
11     (txList, sigList, txHash) = decodeTransaction(txBytes);
```

Fig. 9: A *hash collisions with multiple variable length arguments* bug injected by *HuangGai*

4.5 RQ3: Analysis tool based evaluation

We use the aforementioned 6 analysis tools to detect the bugs injected by *HuangGai* and *SolidiFI*. The following formula is to calculate the ratio of bugs detected to bugs injected:

$$captureRate = BDN \div BAN \tag{3}$$

where *BDN* represents the number of bugs detected by an analysis tool, and *BAN* represents the number of bugs that can be activated injected by the bug injection tool. A lower *captureRate* indicates the weaker ability of an analysis tool to detect injected bugs. Since the output of an analysis tools is usually the types and locations of bugs (in the form of line numbers), we employ *line matching* as the detection criterion, i.e., when the bug type and location (line number) reported by the analysis tool match the type and location of the injected bug, we will count the injected bug as a detected bug. We manually check the tools' documents to map the bug types that these tools can detect to the bug types that *HuangGai* and *SolidiFI* can inject. We install the latest versions of the analysis tools and set the timeout value for each tool to 15 mins per contract and bug type. It is worth noting that the original bugs in the *injected contracts*, i.e., those not generated by the injection tools, do not affect calculating the *captureRate*.

Result. In this experiment, we use the original version of the dataset. *SolidiFI* can only provide *loc* and *length* of its injected code snippet other than the exact locations (i.e., line numbers) of the injected bugs. If we regard *loc* as the location of the injected bug, the *captureRate*(s) of *SolidiFI* is **0%** based on the detection criterion of *line matching*. This is because there is no bug inserted at *loc* (e.g., as shown in Fig 1). To obtain the true *captureRate*(s) of *SolidiFI*, we adjust its detection criterion to *range matching*. If the bug location reported by the analysis tool is in the code snippet inserted by *SolidiFI* (the code snippet range is calculated by *loc* and *length*), and the reported bug type matches the injected bug type, we count the injected bug as a detected bug.

Table 4 shows the *captureRate*(s) of *HuangGai* (based on the detection criterion of *line matching*) and *SolidiFI* (based on the detection criterion of *range matching*). In Table 4, we omit the *captureRate(s)* of a bug injection tool in two cases: 1) This bug injection tool is not designed to inject a type of bugs. 2) A type of bugs injected by this bug injection tool cannot be activated. The experimental results of *realRate(s)* show which bug injection tool will have the above two cases when injecting which type of bugs. * represents that an analysis tool is not designed to detect a type of bug. It can be seen from Table 4 that compared to the bugs injected by *HuangGai*, the bugs injected by *SolidiFI* are easier to be detected by the detection tools. We randomly select 20% of *injected contracts*, use the detection tools to detect these contracts again, and manually track the detection situation. We found that *Slither* is the best performing tool. It reports most of the bugs injected by *SolidiFI* and some bugs injected by *HuangGai* (however, it also produces a lot of false positives). *Mythril*, *Manticore*, *Securify* exceed the timeout value when analyzing many contracts, however, when they do not time out, they detect most of the bugs injected by *SolidiFI* and some bugs injected by *HuangGai*. The experimental results of *captureRate(s)* also mean that the use of *HuangGai* to evaluate detection tools can reveal more weaknesses of detection tools.

Analysis. We describe the performance of each detection tool in detecting injected bugs and point out possible directions for improvement.

The bug detection tools based on pattern matching or feature capture, represented by *SmartCheck* and *Slither*, show

TABLE 4: The *captureRate(s)* of analysis tools when detecting bugs injected by *HuangGai(HG)* and *SolidiFI(SF)*

		captureRate					
Bug type	Injection tool	*SmartCheck*	*Slither*	*Mythril*	*Manticore*	*Maian*	*Securify*
Transaction order dependence	*HG*	0.0%	*	*	*	*	0.0%
	SF	0.0%	*	*	*	*	4.8%
Results of contract execution affected by miners	*HG*	7.4%	35.2%	1.8%	0.1%	*	*
	SF	55.8%	100.0%	5.4%	8.9%	*	*
Unhandled exception	*HG*	26.2%	94.4%	0.2%	*	*	7.2%
	SF	25.8%	100.0%	6.7%	*	*	19.4%
Integer overflow and underflow	*HG*	*	*	1.0%	0.8%	*	*
	SF	*	*	15.6%	10.2%	*	*
Use *tx.origin* for authentication	*HG*	69.8%	72.6%	*	*	*	*
	SF	70.6%	95.6%	*	*	*	*
Re-entrancy	*HG*	*	100.0%	14.5%	0.0%	*	30.4%
Wasteful contracts	*HG*	*	63.1%	0.7%	*	0.0%	8.2%
	SF	*	98.4%	27.7%	*	0.0%	20.4%
Short address attack	*HG*	*	*	*	*	*	*
Suicide contracts	*HG*	*	76.3%	13.4%	0.0%	0.0%	*
Locked ether	*HG*	60.0%	78.0%	*	*	0.0%	0.0%
Forced to receive ether	*HG*	85.7%	99.4%	*	0.2%	*	0.2%
Pre-sent ether	*HG*	80.1%	99.4%	*	0.0%	*	0.0%
Uninitialized local/state variables	*HG*	*	71.9%	*	0.0%	*	*
Hash collisions with multiple variable length arguments	*HG*	*	*	*	*	*	*
Specify *function* variable as any type	*HG*	*	*	0.0%	*	*	*
Dos by complex *fallback* function	*HG*	*	*	*	*	*	*
Public function that could be declared *external*	*HG*	0.0%	77.6%	*	*	*	*
Non-public variables are accessed by *public*/*external*	*HG*	*	*	*	*	*	*
Nonstandard naming	*HG*	*	82.2%	*	*	*	*
Unlimited compiler versions	*HG*	100.0%	58.2%	*	*	*	0.2%

better detection ability when detecting the bugs injected by *HuangGai*. This is because tools based on pattern matching or feature capture tend to have faster detection speed (so they hardly time out) and often generate a lot of alerts. However, such tools tend to generate a large number of false positives, which reduces the guiding value of the detect results they generate.

The bug detection tools based on symbolic-execution, represented by *Mythril* and *Maticore*, show **weaker** bug detection ability. This is because these tools exceed the timeout value when analyzing many contracts. Although we set a long timeout value (15 mins), symbolic-execution tools need to take a much longer time to cover all the paths of the contracts due to the high complexity of the *injected contracts* generated by *HuangGai*. Besides, we found that even in the case of no timeout, it is difficult for these tools to find the complex bugs (e.g, the *re-entrancy* bugs caused by the interaction of multiple functions in multiple contracts). Our evaluation results are consistent with the evaluation results of existing detection tools [21], [44], [45], i.e., the performance of current bug detection tools is far from meeting the security requirements of smart contracts.

Based on our evaluation results, we believe that an ideal detection tool should have at least two modules: the front-end fast scanning module (e.g, pattern matching or feature capture) module and back-end in-depth analysis module (e.g, symbol recognition or fuzzing). The front-end module should obtain information such as what types of bugs may be contained in the contract, and collect enough auxiliary information (e.g, path pruning conditions) for the back-end module. And the back-end in-depth analysis module should narrow the search range (for symbolic execution) or input space (for fuzzing) based on the information provided by the front-end module, and improve the accuracy as much as possible to obtain ideal detection performance.

It is noteworthy that there are still 4 of the 20 types of bugs injected by *HuangGai* that are not covered by any analysis tool.

4.6 RQ4: Bug injection efficiency

We measure the injection time of *HuangGai* and *SolidiFI* when constructing the evaluation dataset. We calculate the bug injection speed of *HuangGai* and *SolidiFI* as follows:

$$speed = IT \div BIN \tag{4}$$

where *IT* represents the time it takes for the bug injection tool to inject bugs, and *BIN* represents the number of bugs injected by the bug injection tool. Note that we only count the aggregated running time of *ContractExtractor* and *BugInjector* as *IT* rather than that of *ContractSpider* for *HuangGai*.

Result. Table 3 shows the bug injection *speed*(s) of *HuangGai* and *SolidiFI* (in seconds per bug). It can be seen that the bug injection speed of *HuangGai* generally **lags behind** that of *SolidiFI*.

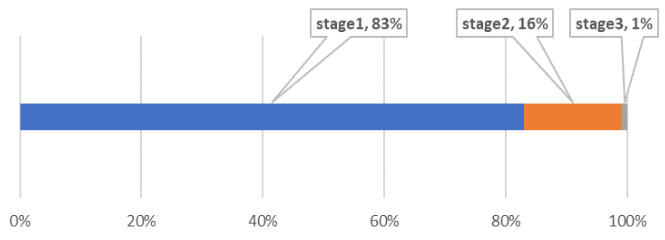


Fig. 10: The proportion of the time for each stage of the bug injection process

Analysis. We analyze the rationale for the lower bug injection speed of *HuangGai*. There are two reasons: 1) *HuangGai* spends substantial time running *solc* and *Slither*. It needs to go through three stages to inject a certain type of bugs into a contract as aforementioned. Running *solc* and *Slither* to generate auxiliary information is the *first stage*. Constructing the contract's control and data flows and ascertaining whether the contract has a basis for injecting a certain type of bugs is the *second stage*. Injecting a certain type of bugs into the contract is the *third stage*. We measure the proportion of time taken for each stage. The results are shown in Fig 10. It can be seen that the running time of *solc* and *Slither* accounts for **most** of the running time of *HuangGai* (83%). 2) Contracts suitable for injecting different types of bugs have different scarcity levels in Ethereum. For instance, *HuangGai* spends substantial time to inject a *specify function variable as any type* bug. This is because *HuangGai* can inject this type of bugs into a contract, only if the contract contains *function* type variables. However, developers rarely use *function* type variables in contracts. According to the statistics, *HuangGai* needs to extract average 21,603 contracts to find a contract suitable for injecting this type of bugs. In contrast, *HuangGai* only takes 209.8 seconds to inject a *suicide contracts* bug. This is because *self-destruct-statements* are more common than *function* type variables. *HuangGai* only needs to inject *suicide contracts* bugs by invalidating the security measures of *self-destruct-statements*.

5 Threats to Validity

We discuss the threats to the validity of *HuangGai* from multiple aspects.

The potential threats to **internal validity** originate from the following two aspects:

- *Restricted extensibility*. So far we can employ *HuangGai* to inject 20 types of bugs into smart contracts. The implementation of *HuangGai* relies on researchers' in-depth understanding of the smart contract bugs. This however restricts users from expanding *HuangGai* to inject new types of bugs. To alleviate this threat, we make *HuangGai*'s source code publicly accessible so that users can learn about the technical details and make extensions from it. We also plan to design machine learning techniques for new bug type learning and injection in the future.
- *Limited injected contract generation capability*. *HuangGai* only injects a certain type of bugs into a contract when the contract has the basis for injecting this type of bug, which means that *HuangGai* cannot inject any type of bugs into any contract. This is a trade-off we make between *injected contract* authenticity and *injected contract* generation capacity. Because as long as there are enough *real contracts* and long enough running time, *HuangGai* can generate enough *injected contracts*. In contrast, there are currently no clear external measures that can improve the *injected contract* authenticity.

The potential threat of **external validity** results from the *tools that HuangGai relies on*. *HuangGai* relies on two open-source tools (i.e., *solc* and *Slither*) to construct the contract's control and data flows. Once these tools are abnormal, it may also affect the validity of *HuangGai*. Nevertheless, *solc* and *Slither* are currently widely-used open-source tools, which are maintained by dedicated organizations and developers, and this will reduce the impact of this threat.

The potential threats of **construct validity** derive from the *repeatability of the evaluation*. To verify the accuracy and authenticity of the bugs injected by *HuangGai*, six smart contract debugging experts participated in our evaluation and manually identified bugs. This may raise the bar for other researchers to re-conduct our evaluation. To alleviate this threat, we provide the evaluation datasets, this allows researchers to completely repeat our evaluation. Besides, we also provide *HuangGai*'s docker image enabling researchers to easily use *HuangGai* for new dataset generation.

Another potential threat to the validity is *timeout value*. In section 4.5, in order to keep the time-consuming evaluation within an acceptable range, if the time taken by an analysis tool exceeds our preset timeout value, we will terminate the execution of the tool and regard as an analysis failure in our evaluation. The setting of the timeout value might cause our evaluation results biased. To reduce the impact of this threat, we refer to the work in [21], [27] to learn the average running time of these tools. Accordingly we set 15 minutes as the timeout value.

6 Related work

6.1 Bug injection tools

Some researchers develop bug injection tools to build large-scale vulnerable program datasets. Bonett et al. propose *μSE* [46], a mutation-based Android static analysis tool evaluation framework. It systematically evaluates Android static analysis tools through mutation analysis to detect weaknesses of these tools. Their work validates the role of bug injection tools in finding weaknesses in analysis tools. Pewny et al. propose *EvilCoder* [47], a bug injection tool that automatically finds the locations of potentially vulnerable source code. It modifies the source code and outputs the actual vulnerabilities. *EvilCoder* first employs automated program analysis technologies to find functions for bug injection. And then, it conducts possible attacks by inserting statements or invalidating security measures. Our work is inspired by *EvilCoder*. Dolan-Gavitt et al. propose *LAVA* [48], a bug injection tool based on dynamic taint analysis. *LAVA* can quickly inject a large number of bugs into programs to build a large-scale corpus of vulnerable programs. In addition, *LAVA* can also provide an input for each injected bug to trigger the bug. Ghaleb et al. propose *SolidiFI* [27], which is the first bug injection tool for Ethereum smart contracts. *SolidiFI* injects bugs into contracts by injecting code snippets containing bugs into all possible locations in the contracts. They employ *SolidiFI* to inject 9,369 bugs of 7 types into 50 contracts. These contracts are then used to evaluate 6 analysis tools. The evaluation results show that these tools cause a large number of false positives and false negatives. Hajdu et al. [26] use software-implemented bug injection to evaluate the behavior of buggy smart contracts, and analyze the effectiveness of formal verification and runtime protection mechanisms in detecting injected bugs. However, these work except for [27] cannot inject bugs into Ethereum smart contracts, or does not provide useable open-source tools.

6.2 Evaluating smart contract analysis tools

Some studies are devoted to evaluating the detection abilities of smart contract analysis tools. Zhang et al. [28] propose an Ethereum smart contract bug classification framework and construct a dataset for this framework. They utilize the constructed dataset to evaluate 9 analysis tools and obtain some interesting findings. Chen et al. [49] evaluate the performance of 6 analysis tools to identify smart contract control flow transfer. They find that these tools cannot identify all control flow transfers. To solve this problem, they propose a more effective control flow transfer tracing approach to reduce the false negatives of analysis tools. Durieux et al. [21] conduct a large-scale evaluation of 9 analysis tools upon 47,587 contracts. They find that these tools would produce a large number of false positives and false negatives. In addition, they present *SmartBugs* [50], an execution framework that integrates 10 analysis tools. Parizi et al. [44] evaluate 4 analysis tools using several handwritten datasets. They think that *SmartCheck* can detect the most bugs, and *Mythril* has the highest accuracy. The existing evaluations of Ethereum smart contract analysis tools rely on either small-scale labelled handwritten datasets or unlabelled *real contract* datasets. This makes it impossible to effectively and precisely evaluate the real performance of analysis tools on bug detection. The emergence of *HuangGai* is expected to solve this dilemma.

6.3 Ethereum buggy smart contract datasets

Some organizations and researchers provide buggy Ethereum smart contract datasets to show developers examples of various bugs and provide benchmarks for smart contract analysis tool evaluation. *SmartContractSecurity* [23] provides a list of 36 types of Ethereum smart contract bugs and creates the exemplary buggy contracts for each type of bug. *Crytic* [24] provides a buggy contract dataset covering 12 types of common Ethereum security issues. However, most of the contracts in the dataset have not been updated in the last two years. Zhang et al. [28] provide a buggy contract dataset covering 49 types of smart contract bugs. This dataset is currently the largest handwritten dataset in terms of the number (173) of contracts. Durieux et al. [21] create two datasets. One contains 47,398 unlabeled *real contracts*. The other comprises 69 labeled buggy *handwritten contracts*. According to the smart contract bug classification scheme provided by *DASP* [51], they classify the bugs in 69 contracts into 10 types. The labeled buggy contract datasets provided by the above work all share the following limitations: inadequate number of contracts, small contract code size, and lack of real business logic in contracts.

6.4 Exploiting smart contract bugs

Some studies are performed to find and exploit bugs in smart contract. Jiang et al. [6] propose *ContractFuzzer*, a smart contract fuzzing tool. *ContractFuzzer* generates fuzzing inputs and uses these inputs to run smart contracts to trigger bugs in smart contracts. Zhang et al. propose *ETPLOIT* [52], a fuzzing-based smart contract bug exploit generator. *ETHPLOIT* employs static taint analysis to generate transaction sequences that can trigger bugs. It adopts a dynamic seed strategy to overcome hard constraints in the execution paths. Feng et al. propose *SMARTSCOPY* [53], an automatic generation tool for adversarial contracts. *SMARTSCOPY* adopts techniques such as symbolic execution to identify and exploit bugs in the victim's smart contracts. Krupp et al. [54] provide the definitions of vulnerable contracts. They present *TEETHER*, which can generate exploits by analyzing the bytecode of the contracts.

7 Conclusion and future work

We propose a new approach, *HuangGai*, to automatically inject 20 types of bugs into Ethereum smart contracts. We implement a bug injection tool, based on this approach. Moreover, we conduct large-scale experiments to evaluate *HuangGai*, and the experimental results show that it can inject more types of bugs than state-of-the-art tools with higher accuracy and authenticity, and label bug locations more precisely. We also select 6 widely used analysis tools to detect the bugs injected by *HuangGai*. The experimental results show that these tools cannot effectively detect most of the bugs injected by *HuangGai*. That is, the buggy contracts generated by *HuangGai* can better uncover the weaknesses of these tools. Finally, we use *HuangGai* to construct 3 *real contract* datasets.

For future work, first, we plan to study how to improve *HuangGai* into a flexible and extensible bug injection framework. i.e., users only need to define some rules, and then *HuangGai* just can inject a new type of bugs. After solving the extensibility limitation, we will try to expand *HuangGai* to enable it to inject more types of bugs. Besides, we will also optimize *HuangGai* so that make it can adapt to the latest Solidity version.

Acknowledgments

The work is supported by the National Natural Science Foundation of China under Grant Nos. 61572171, 61761136003, the Fundamental Research Funds for the Central Universities under grant No.B210203033, B210202075, the Natural Science Foundation of Jiangsu Province under Grant No.BK20191297.

REFERENCES

[1] W. Zou, D. Lo, P. S. Kochhar, X. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart contract development: Challenges and opportunities," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[2] V. Buterin *et al.*, "Ethereum white paper," *GitHub repository*, pp. 22–23, 2013.

[3] C. Diligence. Ethereum smart contract security best practices. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/

[4] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1503–1520.

[5] I. Grishchenko, M. Maffei, and C. Schneidewind, "Ethertrust: Sound static analysis of ethereum bytecode," *Technische Universität Wien, Tech. Rep*, 2018.

[6] 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.

[7] 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.

[8] 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.

[9] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 442–446.

[10] 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.

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

[12] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 65–68.

[13] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[14] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2018, pp. 9–16.

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

[16] 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.

[17] 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.

[18] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 778–788.

[19] J. He, M. Balunovi, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *the 2019 ACM SIGSAC Conference*, 2019.

[20] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1661–1677.

[21] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 530–541.

[22] SMARX. (2021, Mar.) Warmup. [Online]. Available: https://capturetheether.com/challenges/

[23] SmartContractSecurity. (2021, Jan.) Smart contract weakness classification and test cases. [Online]. Available: https://swcregistry.io/

[24] T. of Bits. (2021, Jan.) Examples of solidity security issues. [Online]. Available: https://github.com/crytic/not-so-smart-contracts

[25] OpenZeppelin. (2021, Jan.) Web3/solidity based wargame. [Online]. Available: https://ethernaut.openzeppelin.com/

[26] Á. Hajdu, N. Ivaki, I. Kocsis, A. Klenik, L. Gönczy, N. Laranjeiro, H. Madeira, and A. Pataricza, "Using fault injection to assess blockchain systems in presence of faulty smart contracts," *IEEE Access*, vol. 8, pp. 190 760–190 783, 2020.

[27] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 415–427. [Online]. Available: https://doi.org/10.1145/3395363.3397385

[28] P. Zhang, F. Xiao, and X. Luo, "A framework and dataset for bugs in ethereum smart contracts," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 139–150.

[29] smartdec. (2020, Apr.) a static analysis tool that detects vulnerabilities and bugs in solidity programs. [Online]. Available: https://tool.smartdec.net/

[30] crytic. (2020, Apr.) Static analyzer for solidity. [Online]. Available: https://github.com/crytic/slither

[31] ConsenSys. (2020, Jan.) Security analysis tool for evm bytecode. [Online]. Available: https://github.com/ConsenSys/mythril

[32] trailofbits. (2020, Apr.) Symbolic execution tool. [Online]. Available: https://github.com/trailofbits/manticore

[33] MAIAN-tool. (2020, Apr.) Maian: automatic tool for finding trace vulnerabilities in ethereum smart contracts. [Online]. Available: https://github.com/MAIAN-tool/MAIAN

[34] ChainSecurity. (2020, Apr.) A security scanner for ethereum smart contracts. [Online]. Available: https://securify.chainsecurity.com/

[35] Ethereum. (2020) The development documents of solidity. [Online]. Available: https://docs.soliditylang.org/en/v0.8.0/

[36] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 79–94.

[37] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[38] I. Group *et al.*, "1044-2009-ieee standard classification for software anomalies," *IEEE, New York*, 2010.

[39] Scrapy. (2020) Scrapy, a fast high-level web crawling and scraping framework for python. [Online]. Available: https://github.com/scrapy/scrapy

[40] Ethereum. (2020) Javascript bindings for the solidity compiler. [Online]. Available: https://github.com/ethereum/solc-js

[41] S. Jiang, J. Chen, Y. Zhang, J. Qian, R. Wang, and M. Xue, "Evolutionary approach to generating test data for data flow test," *IET Software*, vol. 12, no. 4, pp. 318–323, 2018.

[42] G. Inc. (2020) Open-source project repository. [Online]. Available: https://github.com/

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

[44] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," in *Proceedings of the 28th Annual*

International Conference on Computer Science and Software Engineering, ser. CASCON '18. USA: IBM Corp., 2018, p. 103–113.

[45] A. Ghaleb and K. PattabiramanD, "How effective are smart contract analysis tools ? evaluating smart contract static analysis tools using bug injection," *2020 ACM 29th International Symposium on Software Testing and Analysis (ISSTA)*, 2020.

[46] R. Bonett, K. Kafle, K. Moran, A. Nadkarni, and D. Poshyvanyk, "Discovering flaws in security-focused static analysis tools for android using systematic mutation," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18. USA: USENIX Association, 2018, p. 1263–1280.

[47] J. Pewny and T. Holz, "Evilcoder: Automated bug insertion," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ser. ACSAC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 214–225. [Online]. Available: https://doi.org/10.1145/2991079.2991103

[48] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 110–121.

[49] 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)*, 2019, pp. 1–11.

[50] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: A framework to analyze solidity smart contracts," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 1349–1352.

[51] N. Group. Decentralized application security project. [Online]. Available: https://dasp.co/

[52] Q. Zhang, Y. Wang, J. Li, and S. Ma, "Ethploit: From fuzzing to efficient exploit generation against smart contracts," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 116–126.

[53] R. B. Yu Feng, Emina Torlak, "Precise attack synthesis for smart contracts," *arXiv preprint arXiv:1902.06067*, 2019.

[54] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1317–1333.