HuangGai: An Ethereum Smart Contract Bug Injection Framework

Anonymous Author(s)

ABSTRACT

Although many smart contract analysis tools have been proposed to detect bugs in smart contracts, the evaluation of analysis tools has always been troubled by the lack of the buggy *real contract* (i.e., smart contracts deployed on Ethereum) datasets, which makes the evaluation unable to show the real performance of the analysis tools. An effective way to solve this problem is to inject bugs into the *real contracts* and automatically label the locations and types of the injected bugs. Although Ghaleb et al. propose the first tool, *SolidiFI*, to automatically inject bugs into Ethereum smart contracts, *SolidiFI* has defects in many aspects (eg., only 7 types of bugs can be injected, the injection accuracy is low, and the locations of the injected bugs cannot be accurately labeled). To solve the above limitations, we design an approach to automatically inject bugs into Ethereum smart contracts, and based on our approach, we implement an open-source tool, *HuangGai*. *HuangGai* first collects smart contracts deployed on Ethereum, and then injects 20 types of bugs into the contracts by analyzing the contracts' control flow and data flow. Experimental evaluations show that compared to *SolidiFI*, *HuangGai* can inject more types of bugs more accurately, and the existing analysis tools can only detect part bugs injected by *HuangGai*. Our work forms the basis of an approach for generating large-scale buggy contract datasets. We hope that through *HuangGai*, users can generate buggy contract datasets to support the research of analysis tools and enable the evaluation show the true performance of the analysis tools.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Ethereum, Solidity, Smart contract security, Bug injection

ACM Reference Format:

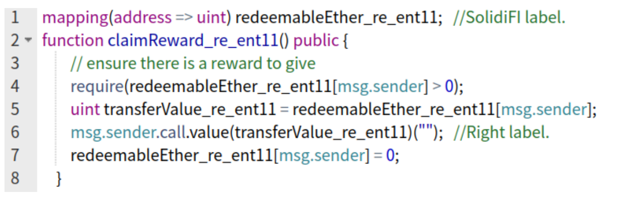
Anonymous Author(s). 2021. HuangGai: An Ethereum Smart Contract Bug Injection Framework. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Smart contracts are autonomous programs running on the blockchain [44]. Ethereum is currently the world's largest platform that supports smart contracts [2], it provides programming languages to developing contracts and allows developers to deploy contracts on Ethereum. Similar to traditional computer programs, it is difficult to avoid bugs in contracts, and the deployed contracts cannot be modified for patching the bugs. Therefore, it is important to detect and fix all bugs in the contracts before deploying them.

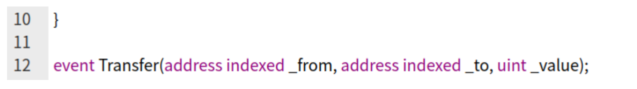
Although recent studies have proposed many analysis tools for detecting bugs in contracts [4, 5, 7, 8, 14, 17, 19, 21, 23, 24, 27, 36, 37, 39], when users evaluate analysis tools, they are always faced with the problem of lacking large-scale contract datasets. The current evaluation mainly uses handwritten datasets [12, 28, 29, 33, 35] (i.e., these datasets composed of buggy contracts manually developed by datasets authors based on bugs' characteristics) to test detection abilities of the analysis tools. However, these handwritten datasets generally have the following limitations:

- *Handwritten contracts lack real business logic.* The purpose of datasets authors developing these contracts is to make bugs. Compared with *real contracts*, handwritten contracts lack real business logic.
- *The code size of handwritten contracts is small.* We count handwritten contracts (from 5 widely used handwritten datasets [12, 28, 29, 33, 35]) and 66,205 *real contracts*. Results show that the average number of *loc* (line of code) of handwritten contracts is 42, while the average number of *loc* of real contracts is 432.
- *The number of handwritten contracts is small.* Handwritten datasets often contain only about 100 contracts. When it comes to a certain type of bug, only a few contracts contain this type of bug.



```
mapping(address => uint) redeemableEther_re_ent11;  //SolidiFI label.
function claimReward_re_ent11() public {
    // ensure there is a reward to give
    require(redeemableEther_re_ent11[msg.sender] > 0);
    uint transferValue_re_ent11 = redeemableEther_re_ent11[msg.sender];
    msg.sender.call.value(transferValue_re_ent11)("");  //Right label.
    redeemableEther_re_ent11[msg.sender] = 0;
}
```

Figure 1: *SolidiFI*'s label error



```
}

event Transfer(address indexed _from, address indexed _to, uint _value);
```

Figure 2: Another type of *SolidiFI*'s label error

An effective way to solve the above limitations is to inject bugs into the *real contracts* and automatically label the locations and types of the injected bugs. Ghaleb et al. [16] propose the first Ethereum smart contract bug injection tool, *SolidiFI*. *SolidiFI* injects bugs by inserting buggy code snippets (i.e., code snippets containing bugs) into all potential locations in a contract. However, *SolidiFI* still has limitations in many aspects:

- *Most of the bugs injected by SolidiFI are independent of the contract's original control flow and data flow. SolidiFI* injects bugs by directly inserting buggy code snippets prepared in advance into the contracts, without using any structure existing in the contracts, which makes bugs injected by *SolidiFI* are independent of the contrac's original control flow and data flow.
- *SolidiFI can only inject 7 types of bugs*. However, existing study [42] shows that there are far more than 7 types of Ethereum smart contract bugs.
- *SolidiFI cannot accurately inject bugs.* Part of the bugs injected by *SolidiFI* cannot be exploited by external attackers. eg., in the contract that contains the code snippet of Fig 1, *SolidiFI* did not insert any statement that increases the value of the variable *redeemableEther_re_ent11* (declared in line 1), which makes the statement in line 6 dead code and also means that this *re-entrancy* bug does not exist.
- *SolidiFI cannot label bug locations correctly.* Fig 1 shows a code snippet inserted by *SolidiFI*, which *SolidiFI* claims this code snippet contains a *re-entrancy* bug. *SolidiFI* labels the *re-entrancy* bug location on line 1. However, most analysis tools determine that line 6 caused this bug instead of line 1. The reason for this situation is that *SolidiFI* labels the code snippet's 1st line as causing the bug. Besides, *SolidiFI* also has another type of labeling error. As shown in Fig 2, *SolidiFI* labels the line 10 as causing a *re-entrancy* bug, but the line 10 does not contain statement.

To solve the above limitations, we propose *HuangGai*[1], an Ethereum smart contract bug injection framework. *HuangGai* first collects *real contracts* and then constructs these contract's data flow and control flow. By analyzing contracts' data flow and control flow, *HuangGai* extracts contracts suitable for injecting bugs and obtains the data required for injection. Finally, by inserting code into the contracts or invalidating the contracts' security measures, *HuangGai* can inject 20 types of bugs into smart contracts.

In summary, we make the following contributions:

(1) We design an approach to automatically inject bugs into Ethereum smart contracts, and based on our approach, we implement an open-source tool, *HuangGai*, which can inject 20 types of bugs into contracts. It is worth noting that among the 20 types of bugs injected by *HuangGai*, 4 types are still not covered by any current analysis tool. The emergence of buggy contracts containing these 4 types of bugs is expected to help researchers develop analysis tools that can detect these 4 types of bugs.

(2) We conduct extensive experiments to evaluate *HuangGai*. Experimental results show that compared to *SolidiFI*, *HuangGai* can inject more types of bugs more accurately.

(3) We use the state-of-the-art analysis tools [3, 9, 10, 25, 34, 38] to detect buggy contracts generated by *HuangGai*. The detection results show that under the premise that *HuangGai* and these analysis tools use the same bug labeling criteria, these analysis tools cannot detect most of the bugs injected by *HuangGai*.

(4) Through *HuangGai*, we generate and release the following 3 datasets:
   - *Dataset 1*[2]: This dataset consists of 964 buggy contracts, covering 20 types of bugs, and 3 researchers who familiar with smart contract bugs check the injected bugs in these buggy contracts to ensure that all injected bugs can be activated (i.e., the injected bugs can be exploited by external attackers). As far as we know, *dataset 1* is currently the largest (number of contracts) buggy contract dataset with bug labels.
   - *Dataset 2*[3]: This dataset consists of 4,744 buggy contracts, covering 20 types of bugs. Users can use the contracts in *dataset 1 and 2* as the benchmark to evaluate the performance of analysis tools, to obtain the true performance of the analysis tools.
   - *Dataset 3*[4]: This dataset consists of 66,205 *real contracts*. Users can analyze the contracts in this dataset to know the current overview of Ethereum smart contracts.

The rest of this paper is organized as follows: Section 2 introduces the necessary background. Section 3 presents *HuangGai*. In Section 4, we describe the evaluation results of *HuangGai*. After introducing the related work in Section 5, we conclude the paper with future work in Section 6.

2 BACKGROUND

2.1 Ethereum smart contract

Ethereum [2] provides a variety of programming languages for developers to develop smart contracts. Users deploy the contracts to Ethereum by sending transactions. In Ethereum, each contract or user is assigned a unique address as identification. Ether is the cryptocurrency used by Ethereum, and both contracts and users can trade Ethers. To avoid abusing Ethereum's computational resources, Ethereum charges fees (called *gas*) for each executed contract statement.

2.2 Solidity

Solidity [13] is currently the most mature and widely used Ethereum smart contract programming language. Solidity has Turing-complete expression ability and can express arbitrary complex logic. Users can use Solidity to develop contracts, and then use the compiler to generate the contracts' bytecode. Solidity is a fast-evolving language, and new versions of Solidity containing breaking changes are released every few months. When developing a contract, developers need to specify the Solidity version used for development, so as to use the compiler of the corresponding Solidity version to compile the contract. Solidity assigns a *function selector* to each function, and the overridden function has the same function selector value as the overriding function. Solidity supports (multiple) inheritance including polymorphism and and Solidity will specify the linear inheritance order from the base contract to the derived contract (even if multiple inheritance occurs). Solidity provides *require-statement*

[1] *HuangGai* is available at https://github.com/xf97/HuangGai.

[2] https://github.com/xf97/HuangGai/tree/master/manualCheckDataset
[3] https://github.com/xf97/HuangGai/tree/master/injectedContractDataSet
[4] https://github.com/xf97/HuangGai/blob/master/sourceCodeDateSet.zip

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.

2.3 Smart contract analysis tool

Smart contract analysis tools are designed to detect bugs in contracts. Generally speaking, the input of the analysis tools is the source code or bytecode of a contract, and the output is the types of bugs in the contract and the locations of the bugs (in the form of line number or line number range). At present, a variety of technologies have been applied to detect smart contract bugs, such as pattern matching [14, 36], symbolic execution [26], and fuzzing [19].

3 *HUANGGAI*

3.1 Overview of *HuangGai*

HuangGai is an Ethereum smart contract bug injection framework, it injects 20 types of bugs into contract source code. The reason why we chose to complete the injection at the source code level instead of bytecode level is that we don't want to exclude analysis tools that can only analyze source code from the influence of *HuangGai* (while the analysis tools that analyze bytecode can generate bytecode for analysis by compiling source code). The definitions, names, and severity levels of 20 types of bugs injected by *HuangGai* come from [42]. In [42], Zhang et al. also proposed a characteristic-based bug detection criteria, i.e., when there are several specific characteristics in a contract, it can be determined that this contract contains a certain type of bug. Therefore, *HuangGai* injects bugs by constructing bugs' characteristics in the contract based on [42]. Currently, *HuangGai* can inject the 20 most dangerous types of bugs in [42]. The workflow of *HuangGai* is shown in Fig 3.

3.2 *ContractSpider*

One of our goals is to enable users to use *HuangGai* to obtain the buggy contract datasets without preparing contracts in advance. Therefore, *HuangGai* first needs to collect *real contracts* as materials for injecting bugs. We implement *ContractSpider* based on the [32], which is a parallel high-performance web crawler. *ContractSpider* collects contract source code by crawling websites that store open-source *real contracts* (eg., http://etherscan.io/), and saves these source code as Solidity files. Note that *ContractSpider* runs independently of *ContractExtractor* and *BugInjector*, that is, users only need to run *ContractSpider* once to collect the real contracts, and then can run *ContractExtractor* and *BugInjector* multiple times to inject bugs into the contracts.

3.3 *ContractExtractor*

Our another goal is to inject bugs into the contract while keeping the original content of the contract as much as possible. To achieve this goal, *HuangGai* need to inject bugs only by slightly modifying the contract, which requires *HuangGai* to determine whether a contract has the basis for injecting a certain type of bug. *ContractExtractor* is designed to achieve this goal. *ContractExtractor* reads the contract collected by *ContractSpider*, constructs the contract's data flow and control flow, and then analyzes the data flow and control flow to determine whether the contract can inject a certain type of bug. If a contract has the basis for injecting a certain type of bugs, then *ContractExtractor* will pass the contract and the data required for injecting bugs to *BugInjector*; otherwise, the *ContractExtractor* will read another contract. It is worth noting that because the characteristics of each type of bug are different, the required basis for injecting each type of bug is not the same, which leads to the *ContractExtractor* of each type of bug is different. Due to space limitations, in this paper, we select 2 representative *ContractExtractors* (i.e., the *ContractExtractors* of the *re-entrancy* bug and the *integer overflow and underflow* bug) as examples to describe how *ContractExtractor* works.

3.3.1 ContractExtractor of re-entrancy (CER). According to [42], the characteristic of the *re-entrancy* bug is **the contract uses the *call-statement* to send ethers and then deduct the number of tokens held by the payee address**. *HuangGai* can insert the *call-statements* for sending ethers into a contract, so the key to injecting *re-entrancy* bugs is to find the statements in the contract that deduct the number of tokens held by the payee address (we call such statements *deduct statements*). In order to find *deduct-statements*, *HuangGai* first needs to find the variables that record the relationship between the addresses and the number of tokens held by the addresses (we call such variables *ledgers*).

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

(1) **Step 1**: *CER* searches for the *deposit paths* in the contract. The *deposit path* refers to a function-call path in the contract with the following conditions: 1) The entry function of this path is a function declared as ***payable***. 2) There is at least 1 value-increased operation on a *mapping(address=>uint256)* variable in this path. We call the *mapping(address=>uint256)* variable *potential ledger*, and the set of all *potential ledgers* in the *deposit paths* the set *PLS*.
(2) **Step 2**: *CER* searches for the *withdrawal paths* in the contract. The *withdrawal path* refers to a function-call path in the contract with the following conditions: 1) In this path, there is at least one value-decreased operation on a *potential ledger*. We call the *potential ledger* that meets this condition the *target ledger*. 2) In this path, there is at least 1 operation to send ethers, where the payee address needs to be the same as the value-decreased address in the *target ledger*. We call the set of all *target ledgers* the set *ledgerSet*, and the set of the locations of all value-decreased operations the set *deductSet*.
(3) **Step 3**: All variables in *ledgerSet* can be regarded as the *ledgers* in the contract and the *deductSet* records the locations of all *deduct-statements*.

If a contract's *deductSet* is not empty, *HuangGai* only needs to insert *call-statements* for sending ethers before all locations in *deductSet* to inject *re-entrancy* bugs into the contract. eg., in the contract shown in Fig 4, there are 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 $\{line17\}$, so *HuangGai* only needs to insert a *call-statement* for sending ethers in line 16 to inject a *re-entrancy* bug.

To construct a contract's *ledgerSet* and *deductSet*, *HuangGai* needs to analyze the contract's data flow and control flow. Specifically, *HuangGai* needs to construct all function-call paths in the

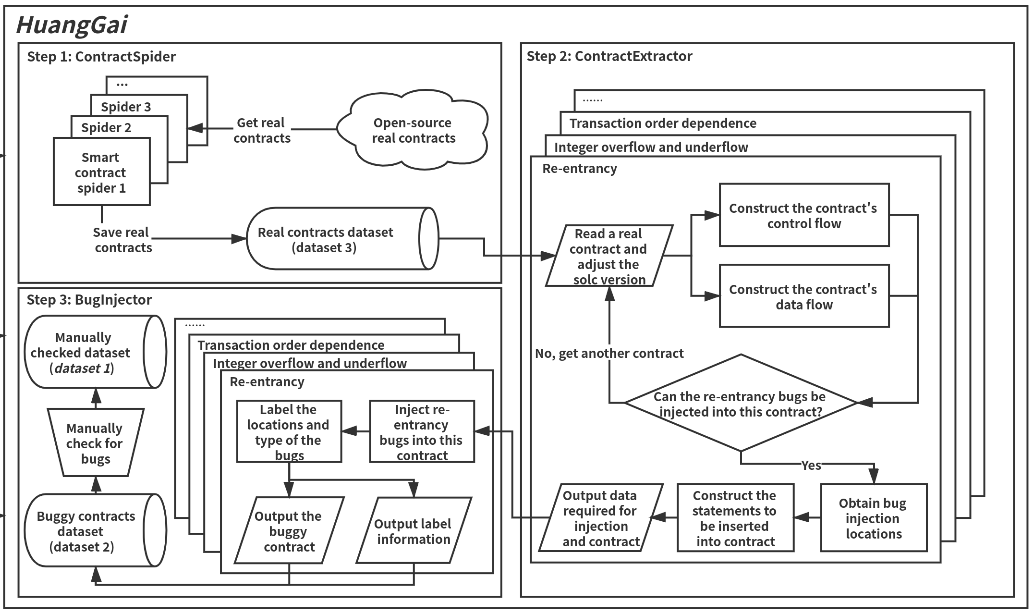
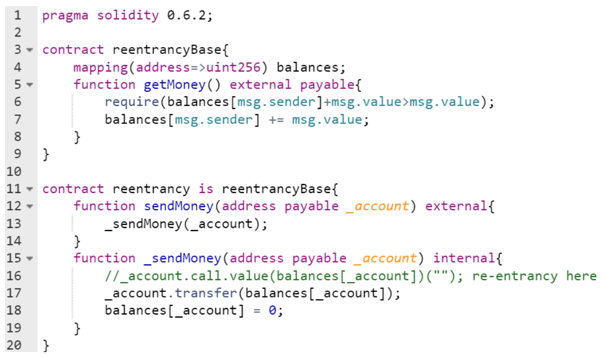

HuangGai
Step 1: ContractSpider
...
Spider 3
Spider 2
Smart contract spider 1
Get real contracts
Open-source real contracts
Save real contracts
Real contracts dataset (dataset 3)
Step 2: ContractExtractor
......
Transaction order dependence
Integer overflow and underflow
Re-entrancy
Read a real contract 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
Ope 1: Start ContractSpider
Users
Ope 2: Specify requirements
Ope 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

Figure 3: *HuangGai* workflow



```
pragma solidity 0.6.2;

contract reentrancyBase{
    mapping(address=>uint256) balances;
    function getMoney() external payable{
        require(balances[msg.sender]+msg.value>msg.value);
        balances[msg.sender] += msg.value;
    }
}

contract reentrancy is reentrancyBase{
    function sendMoney(address payable _account) external{
        _sendMoney(_account);
    }
    function _sendMoney(address payable _account) internal{
        //_account.call.value(balances[_account])(""); re-entrancy here
        _account.transfer(balances[_account]);
        balances[_account] = 0;
    }
}
```

Figure 4: A contract that *HuangGai* can inject a *re-entrancy* bug

contract (even if the contract is a derivative contract), and be able to track the definition and use of data in each path. Based on emph-solc[5] and *Slither*[6], *HuangGai* is able to construct a contract's data flow and control flow (the *ContractExtractors* of other types of bugs also use the same way to construct the contracts' data flow and control flow):

- **Data flow**: Using *solc* to compile a contract can generate the abstract syntax tree (*AST*) of the contract. By analyzing *AST*, *HuangGai* can obtain the following information: the *definition-use* pairs [20] of the data and linear inheritance order. Based on the above information, *HuangGai* can track the definition and use of each data (eg., the *ledger* variables).
- **Control flow**: Based on the control flow graph (*CFG*) of each function and function-call graph generated by *Slither*, combined with the linear inheritance order generated by *solc*, *HuangGai* can generate the (derived) contract's *CFG*. When a function (or modifier) overrides the same function (or modifier) of the base contract, *HuangGai* uses the *function selector* to identify the overridden function in the path and replace it with the overriding function.

Through *solc* and *Slither*, *HuangGai* can construct all the function-call paths and track the definition and use of the data, so as to construct a contract's *ledgerSet* and *deductSet* (algorithm 1 shows the specific process). When a contract's *deductSet* is not empty, *CER* will pass the contract's source code, the payee addresses, and the *deductSet* to the *BugInjector* of *re-entrancy* bug to inject *re-entrancy* bugs.

3.3.2 *ContractExtractor of integer overflow and underfow (CEI).*

According to [42], the characteristics of the *integer overflow and underflow* (*IOA*) bug are:

- *Characteristic 1*: The maximum (or minimum) values 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 with the following conditions in a contract:

- *Condition 1*: The types of the operand variables and the result variable are the same type. This condition can ensure that the statement meets *characteristic 1*.
- *Condition 2*: In the integer arithmetic statement, at least 1 operand variable is a parameter given by the caller. This

[5]Solidity official compiler
[6]A widely used *Solidity* static analysis framework.

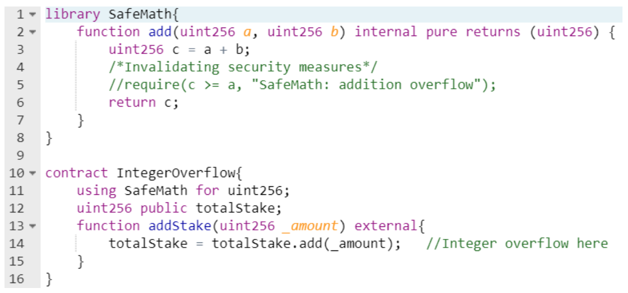
Algorithm 1: *ledgerSet* And *deductSet* Generation Algorithm

```
   Input: Contract's source code SC
   Output: ledgerSet, deductSet
1  changeSolcVersionBySolc(SC);
   // Adjust the local solc version to compile the
      contract.
2  ContractAstSet C = getContractASTBySolc(SC);
3  Set P, PLS;
   // function-call paths set.
4  foreach contractAst in C do
       // from base contract to derived contract
5      callGraphSet = getFuncCallGraphBySlither(SC);
6      CFGSet = getAFuncCFGBySlither(SC);
7      contractPathSet =
        getContractCFG(CFGSet, callGraphSet);
8      funcAndItsSelector =
        getFuncSelectorBySolc(contractAst);
9      P = P ∪ contractPathSet;
       // Add new function-call paths.
10     foreach path in P do
11         foreach (oldFunc, oldSelector) in path do
12             foreach (newFunc, newSelector) in
                funcAndItsSelector do
13                 if newSelector == oldSelector then
                       /* Override                    */
14                     oldFunc is replaced by newFunc;
15                 end
16             end
17         end
18     end
19 end
20 PLS = getPLSSet(P, C);
21 Set ledgerSet, deductSet = getResultSets(PLS, P, C);
22 return ledgerSet, deductSet;
```

condition can ensure that the injected bugs can be exploited by external attackers.

We call the integer arithmetic statements that meet *conditions 1* and *2* as the target statements. As shown in Fig 5, when the *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 *check statements* based on our experience: Generally speaking, *require-statements* or *assert-statements* are used to check the results of integer arithmetic statements. Therefore, when the operands that participating in the integer arithmetic statement are the parameters of a *require-statement* or an *assert-statement*, *CEI* will regard this *require-statement* (or *assert-statement*) as a *check statement*.

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



```
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 }
```

Figure 5: How *HuangGai* invalidates security measures.

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

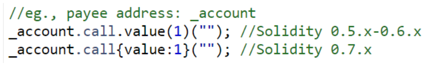
When the *target* set of a contract is not empty, *HuangGai* will be able to inject *IOA* bugs into the contract and pass the contract source code, *target* set, and *opponent* set to *BugInjector*.

3.4 *BugInjector*

BugInjector receives the contract and the data required for injecting bugs passed by the *ContractExtractor*, and then injects bugs into the contract and labels the bugs. According to the data passed by *ContractExtractor*, the *BugInjector* injects and labels bugs by one of the following means for different types of bugs:

- Inserting statements that cause bugs. *BugInjector* inserts the statements that cause the bugs into the contract, and then label the insert locations as existing bugs.
- Invalidating security measures. *BugInjector* first invalidates the security measures in the contract and then labels the statements lacking security protection as existing bugs.

Due to space limitations, we also use the *BugInjectors* of the *re-entrancy* bug and the *integer overflow and underflow* bug as examples to describe how *BugInjector* works. These two *BugInjectors* use different 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
```

Figure 6: How *HuangGai* constructs the *call-statement* based on the payee address.

3.4.1 *BugInjector of re-entrancy (BIR).*

BIR injects *re-entrancy* bugs by inserting statements that cause bugs. The *CER* will pass the following information to *BIR*: the contract source code, the payee addresses, and the locations of the *deduct-statements* (*deductSet*). Based on the above information, *BIR* uses the payee addresses to construct the *call-statements* for sending ethers (as shown in Fig 6), and then inserts the *call-statements* before the *deduct-statements* based on the locations of the *deduct-statements*, and finally labels the lines where the *call-statements* are inserted as existing *re-entrancy* bugs.

3.4.2 *BugInjector of integer overflow and underfow (BII).*

BII injects *IOA* bugs by invalidating security measures. The *CEI* will pass the following information to *BII*: the contract source code, *target* set, and *opponent* set. Among them, the *target* set contains the locations of statements that may cause *IOA* bugs, and the opponent set contains the locations of *check statements*. Therefore, *BII* first invalidates all *check statements* (by changing the statements to comments) in the *opponent* set to make the statements in the *target* set lack security protection, and then labels the statements in the *target* set as existing *IOA* bugs.

4 EVALUATION

We conduct a series of evaluation experiments to answer the following research questions:

- **RQ1**: Compared with similar tools, can *HuangGai* inject bugs more accurately?
- **RQ2**: Compared with similar tools, can users find more defects in analysis tools by using *HuangGai*?
- **RQ3**: Compared with similar tools, how does *HuangGai* perform when injecting bugs?

We design RQ1 to obtain how many bugs can be activated among the bugs injected by *HuangGai*. We design RQ2 to obtain whether the bugs injected by *HuangGai* are more difficult to be detected by analysis tools than bugs injected by similar tools. We design RQ3 to obtain *HuangGai*'s bug injection speed.

4.1 Tools used in the evaluation

In the evaluation, we will use two types of tools: 1) Ethereum smart contract bug injection tool. Such tools are similar tools of *HuangGai*. As far as we know, there is currently only *SolidiFI* a bug injection tool, so we will compare *HuangGai* and *SolidiFI*. 2) Smart contract analysis tools. We will use analysis tools to detect bugs injected by *HuangGai* and *SolidiFI*.

We collect smart contract analysis tools based on the following two channels:

- Analysis tools that have been evaluated in the recent evaluation work [12, 16].
- We use *smart contract security* and *smart contract analysis tools* as keywords to search analysis tools on Github [18], and select the top twenty tools (sorted by the number of *stars* in descending order).

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

- *Criterion 1.* The tool can support the command-line interface. This allows us to automatically run analysis tools to detect bugs.

Table 1: 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*
	criterion 2	*Vandal*, *Echidna*, *MadMax*, *VeriSol*
	criterion 3	*EthIR*, *E-EVM*, *Erays*, *Ethersplay*, *EtherTrust*, *contractLarva*, *FSolidM*, *KEVM*, *SolMet*, *Solhint*, *rattle*, *Solgraph*, *Octopus*, *Porosity*
	criterion 4	*Osiris*, *Oyente*, *HoneyBadger*
Selected	*Maian*, *Manticore*, *Mythril*, *Securify*, *Slither*, *SmartCheck*	

- *Criterion 2.* The input of the tool is the Solidity source code or bytecode.
- *Criterion 3.* The tool is a bug detection tool, i.e., the tool can report the types and locations of bugs in the contracts.
- *Criterion 4.* The tool can detect contracts of Solidity 0.5.0 and subsequent versions. This is because both *HuangGai* and *SolidiFI* are currently only able to inject bugs into contracts of Solidity 0.5.0 and subsequent versions.

According to the collection and selection criteria, we finally select 6 analysis tools to detect bugs injected by *HuangGai* and *SolidiFI*. Table 1 lists the 6 selected tools and excluded tools.

4.2 Dataset for evaluation and environment

As bug injection tools, the output of *HuangGai* and *SolidiFI* are *injected contracts* (i.e., contracts that have been injected bugs by the bug injection tool). Therefore, to compare these two tools, we first need to use *HuangGai* and *SolidiFI* to generate the buggy contract datasets and then evaluate the bug injection abilities of the two tools by inspecting each contract in the buggy contract datasets. To ensure fairness, we use *HuangGai* and *SolidiFI* to inject the same type of bugs into the same contracts to generate the evaluation datasets. It is worth noting that *HuangGai* cannot inject bugs into every contract. Specifically, considering the time and effort constraints, for each type of bug, we first use *HuangGai* to generate 50 *injected contracts* (i.e., contracts that have been injected bugs by the bug injection tool) and then use *SolidiFI* to inject the same type of bugs into the original versions of these *injected contracts*. If *HuangGai* cannot generate 50 contracts that contain a certain type of bug (even if *HuangGai* has scanned the entire real contract dataset), then we will use all *injected contracts* currently generated to build the dataset. Finally, we obtain a dataset consisting of 964 (covering 20 types of bugs) *injected contracts* generated by *HuangGai* and 323 (covering 7 types of bugs) *injected contracts* generated by *SolidiFI*.

Our evaluation environment is a desktop computer running Ubuntu (18.04) operating system, and this desktop computer is equipped with 16GB of memory, the CPU is AMD ryzen5 2600x, and the GPU is NVIDIA GTX 1650.

4.3 RQ1: Compared with similar tools, can *HuangGai* inject bugs more accurately?

We use the following formula to calculate the accuracy of bug injection:

$$accuracyRate = (BIN - IABN) \div BIN \tag{1}$$

In formula 1, *IABN* represents the the number of bugs that cannot be activated, and *BIN* represents the number of bugs injected by the bug injection tool. A bug that can be activated means that the external attacker can exploit the bug. 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 will use *solc* (the same *solc* version as before injecting bugs) to compile the *injected contract*. If there are compilation errors in an *injected contract*, all the injected bugs in the contract are deemed not to be activated.
- Whether the injected bugs can be exploited by external attackers and cause the expected consequences. 3 researchers who familiar with smart contract bugs will manually check each injected bug and reach a consensus through discussion. If an injected bug cannot be exploited by external attackers, the bug will be deemed not to be activated.

However, with *SolidiFI*'s inaccurate bug labeling ability (i.e., in some locations where *SolidiFI* claims to have injected bugs, *SolidiFI* did not insert any code snippet), we cannot find every bug injected by *SolidiFI*. To solve the impact of this problem on the *accuracyRate*(s), we only count the bugs with the correct label locations and manually check each each of them. The correct label location refers to that the code snippets inserted by *SolidiFI* do exist where *SolidiFI* claims to have inserted the code snippets, and the bugs contained in these code snippets are deemed to be labeled correctly.

Results. Table 2 shows the *accuracyRate*(s) of *HuangGai* and *SolidiFI* when injecting various bugs. *N/A* represents that the bug injection tool is not designed to inject this type of bug. It can be seen from Table 2 that compared with *SolidiFI*, *HuangGai* can inject more types of bugs more accurately. Specifically, among the 20 types of bugs, *HuangGai* shows higher *accuracyRate(s)* than *SolidiFI* when injecting 19 types. The *accuracyRate(s)* of *HuangGai* injecting 13 types of bugs reach 100%. When *HuangGai* injects *integer overflow and underflow* bugs, it shows the worst *accuracyRate* among 20 types of bugs, but the *accuracyRate* also reaches 94%.

Analysis. We analyze the reasons that lead to the injection failures of *SolidiFI* and *HuangGai*. The main reasons for the injection failures of *SolidiFI* are: 1) *SolidiFI* is not compatible with Solidity 0.6.0 and subsequent versions, which leads to a large number of compilation errors. Specifically, when using *SolidiFI* to inject bugs into contracts of Solidity 0.6.0 and subsequent versions, due to the grammatical changes between different Solidity versions, there will be a lot of syntax errors in the *injected contracts*. 2) There are problems with the implementation of *SolidiFI*. When *SolidiFI* injects *results of contract execution affected by miners* bugs, *SolidiFI* will insert a piece of text into the contracts, which causes compilation errors when compiling the contracts. 3) Some statements inserted by *SolidiFI* are dead code. When *SolidiFI* injects *re-entrancy* bugs, the statements injected by *SolidiFI* that can cause *re-entrancy* bugs are dead code, so these injected bugs cannot be exploited by external attackers. The main reasons for the injection failures of *HuangGai* are: 1) Compilation errors. Although *HuangGai* are compatible with multiple Solidity versions (0.5.x, 0.6.x, 0.7.x) and only slightly modifies the content of the contract, the statements inserted by *HuangGai* may still cause compilation errors. However, these situations are not common. In the process of compiling all 964 *injected contracts* generated by *HuangGai*, only two compilation errors occur. 2) Excessive security measures. In some contracts, developers use multiple mutually redundant security measures to prevent bugs, resulting in that even if *HuangGai* has invalidated most of the security measures, there are still security measures in the contract to prevent the injected bugs from being activated.

4.4 RQ2: Compared with similar tools, can users find more defects in analysis tools by using *HuangGai*?

We use analysis tools to detect bugs injected by *HuangGai* and *SolidiFI*, and use the following formula to calculate the proportion of bugs detected:

$$captureRate = BDN \div BIN \tag{2}$$

In formula 2, *BDN* represents the number of bugs detected by the analysis tools, and *BIN* represents the number of bugs injected by the bug injection tool. The lower the *captureRate*, the more difficult it is for analysis tools to detect bugs injected by the bug injection tools. The detection criterion we use is *line matching*, 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 10 mins per contract and bug type.

Results. *SolidiFI*'s inaccurate bug labeling ability once again troubled us. When labeling the bug location, *SolidiFI* does not provide the exact location of the injected bug (i.e., line number), but provides two other attributes: *loc* (the line number where the code snippet is inserted) and *length* (the length of the code snippet). If we regard *loc* as the location of the injected bug and based on the detection criterion of *line matching*, since there is no bug at the *loc* location (eg., as shown in Fig 1), the *captureRate*(s) of *SolidiFI* is 0%. This is obviously a distortion of the *captureRate*(s) caused by the wrong label locations. To obtain the true *captureRate*(s) of *SolidiFI*, we adjust the detection criterion to *range matching*, i.e., when 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 also matches the injected bug type, we will count the injected bug as a detected bug. Table 3 shows the *captureRate*(s) of *HuangGai* (based on detection criterion of *line matching*) and *SolidiFI* (based on detection criterion of *range matching*). *N/A* represents that the bug injection tool

Table 2: The *accuracyRate* and *speed* of *HuangGai(HG)* and *SolidiFI(SF)* in injecting various bugs

	accuracyRate		*speed*	
Bug type	*HG*	*SF*	*HG*	*SF*
Transaction order dependence	100.0%	78.5%	33.6	0.6
Results of contract execution affected by miners	100.0%	0.0%	1.0	1.4
Unhandled exception	100.0%	100.0%	3.3	0.6
Integer overflow and underflow	94.1%	86.3%	3.6	1.0
Use *tx.origin* for authentication	100.0%	92.2%	4.5	0.4
Re-entrancy	96.7%	0.0%	334.4	0.4
Wasteful contracts	98.0%	91.7%	4.9	0.7
Short address attack	100.0%	N/A	16.4	N/A
Suicide contracts	100.0%	N/A	210.6	N/A
Locked ether	100.0%	N/A	3.8	N/A
Forced to receive ether	100.0%	N/A	4.9	N/A
Pre-sent ether	100.0%	N/A	4.7	N/A
Uninitialized local/state variables	99.9%	N/A	0.71	N/A
Hash collisions with multiple variable length arguments	100.0%	N/A	84.0	N/A
Specify *function* variable as any type	100.0%	N/A	1948.1	N/A
Dos by complex *fallback* function	100.0%	N/A	60.9	N/A
Public function that could be declared *external*	99.8%	N/A	1.4	N/A
Non-public variables are accessed by *public*/*external* function	99.8%	N/A	0.8	N/A
Nonstandard naming	99.8%	N/A	1.9	N/A
Unlimited compiler versions	100.0%	N/A	6.8	N/A

is not designed to inject this type of bugs. * represents that the analysis tool is not designed to detect this type of bug. # represents that this type of bugs injected by the bug injection tool cannot be activated. It can be seen from Table 3 that even based on the detection criterion of *range matching*, the *captureRate*(s) of *SolidiFI* is still very low. We randomly select 162 (accounting for 50% of the contracts generated by *SolidiFI*) buggy contracts generated by *SolidiFI* from the dataset and manually inspect these contracts. The scope of inspection includes: the contract's source code, the bug label information generated by *SolidiFI* for the contract, and the detection reports generated by the analysis tools for the contract. We find that *SolidiFI* has problems when labeling the locations of the bugs: The *loc* attributes of most bugs are wrong (eg., as shown in Fig 2), which makes it impossible for us to calculate the code snippet range and also leads to the low *captureRate*(s) of *SolidiFI*. Even if we cannot obtain the real *captureRate(s)* of *SolidiFI*, however, the following situation is worthy of our attention: *SolidiFI* injected 1,390 *use* tx.origin *for authentication* bugs and 1,228 *wasteful contracts* bugs into 50 contracts, respectively, and *Slither* also reported 1,390 *use* tx.origin *for authentication* bugs and 1,244 *wasteful contracts* bugs from these contracts. This kind of situation is not uncommon, which may mean that the analysis tools can detect a large number of bugs injected by *SolidiFI*. Besides, it can be seen from Table 3 that these analysis tools cannot detect most of the bugs injected by *HuangGai*, so users can find the defects of the analysis tool by using *HuangGai*.

Analysis. When detecting bugs injected by *HuangGai*, non-symbolic-execution tools (i.e., *SmartCheck* and *Slither*) show better detection abilities. This is because these tools usually use pattern matching to detect bugs, so they are usually able to detect a large number of bugs but bring many false positives. eg., although *Slither* can detect all 24 *re-entrancy* bugs injected by *HuangGai*, it also reports 387 *re-entrancy* bugs in all 23 *injected contracts*. The symbolic-execution tools show poor bug detection abilities. This is because these tools exceed the timeout value when analyzing many contracts. Although we set a long timeout value (10 mins), due to the high complexity of the *injected contracts*, symbolic-execution tools need to take a lot of time to cover all paths of the contracts. Besides, analysis tools can detect part of bugs injected by *HuangGai* reflects a fact, i.e., *HuangGai* has reached a consensus with the developers of the analysis tools in terms of bug labeling criteria. Note that of the 20 types of bugs injected by *HuangGai*, there are still 4 types that are not covered by any analysis tools.

4.5 RQ3: Compared with similar tools, how does *HuangGai* perform when injecting bugs?

When constructing the dataset for evaluation, we count the injection time of *HuangGai* and *SolidiFI*. And use the following formula to calculate the bug injection speed of *HuangGai* and *SolidiFI*:

$$speed = IT \div BIN \qquad (3)$$

In formula 3, *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 for *HuangGai*, we only record the running time of *ContractExtractor* and *BugInjector* as *IT*, not the running time of *ContractSpider*.

Results. Table 2 shows the *speed*(s) of *HuangGai* and *SolidiFI* injecting various bugs, in seconds. It can be seen from Table 2 that the bug injection speed of *HuangGai* generally lags behind that of *SolidiFI*.

Table 3: 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%	*	*	*	*	0.0%
Results of contract execution affected by miners	*HG*	7.1%	33.1%	0.9%	0.1%	*	*
	SF	#	#	#	#	*	*
Unhandled exception	*HG*	27.8%	92.0%	0.0%	*	*	7.4%
	SF	6.1%	0.0%	0.4%	*	*	0.0%
Integer overflow and underflow	*HG*	*	*	0.0%	0.0%	*	*
	SF	*	*	0.0%	0.0%	*	*
Use *tx.origin* for authentication	*HG*	91.7%	71.0%	0.0%	*	*	*
	SF	7.6%	0.0%	0.3%	*	*	*
Re-entrancy	*HG*	*	100%	4.4%	0.0%	*	26.1%
	SF	#	#	#	#	*	#
Wasteful contracts	*HG*	*	62.6%	0.4%	*	0.0%	8.0%
	SF	*	0.0%	0.0%	*	0.0%	0.0%
Short address attack	*HG*	*	*	*	*	*	*
	SF	N/A	N/A	N/A	N/A	N/A	N/A
Suicide contracts	*HG*	*	71.9%	0.0%	0.0%	0.0%	*
	SF	N/A	N/A	N/A	N/A	N/A	N/A
Locked ether	*HG*	60.0%	76.0%	*	*	0.0%	0.0%
	SF	N/A	N/A	N/A	N/A	N/A	N/A
Forced to receive ether	*HG*	85.7%	98.3%	*	0.1%	*	0.0%
	SF	N/A	N/A	N/A	N/A	N/A	N/A
Pre-sent ether	*HG*	84.7%	99.4%	*	0.0%	*	0.0%
	SF	N/A	N/A	N/A	N/A	N/A	N/A
Uninitialized local/state variables	*HG*	*	71.9%	*	0.0%	*	*
	SF	N/A	N/A	N/A	N/A	N/A	N/A
Hash collisions with multiple variable length arguments	*HG*	*	*	*	*	*	*
	SF	N/A	N/A	N/A	N/A	N/A	N/A
Specify *function* variable as any type	*HG*	*	*	0.00%	*	*	*
	SF	N/A	N/A	N/A	N/A	N/A	N/A
Dos by complex *fallback* function	*HG*	*	*	*	*	*	*
	SF	N/A	N/A	N/A	N/A	N/A	N/A
Public function that could be declared *external*	*HG*	0.0%	76.8%	*	*	*	*
	SF	N/A	N/A	N/A	N/A	N/A	N/A
Non-public variables are accessed by *public*/*external*	*HG*	*	*	*	*	*	*
	SF	N/A	N/A	N/A	N/A	N/A	N/A
Nonstandard naming	*HG*	*	84.9%	*	*	*	*
	SF	N/A	N/A	N/A	N/A	N/A	N/A
Unlimited compiler versions	*HG*	100.0%	55.8%	*	*	*	0.0%
	SF	N/A	N/A	N/A	N/A	N/A	N/A

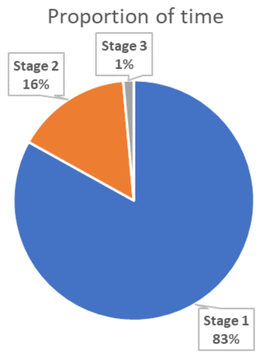


Figure 7: The proportion of the time for each stage of the bug injection process

Analysis. We analyze the reasons for the slow bug injection speed of *HuangGai*. We think there are two reasons: 1) *HuangGai* spends a lot of time running *solc* and *Slither*. *HuangGai* needs to go through the following three stages to inject a certain type of bugs into a contract: run *solc* and *Slither* to generate auxiliary information for constructing the contract's data flow and control flow (*stage 1*), construct the contract's data flow and control flow and determine whether the contract has a basis for injecting a certain type of bugs (*stage 2*), inject a certain type of bugs into the contract (*stage 3*). We count the proportion of time for each stage, and the results are shown in Fig 7. As can be seen from Fig 7, 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. eg., *HuangGai* spends a lot of time to inject a *specify function variable as any type* bug. This is because only when a contract contains the *function* type variables, *HuangGai* will inject the *specify function variable as any type* bugs into the contract by inserting statements that cause bugs. However, developers rarely use *function* type variables in contracts, which makes *HuangGai* need to extract 21,603 (average) contracts to find a contract suitable for injecting this type of bugs. Conversely, *HuangGai* only takes 210.6 seconds to inject a *suicide contracts* bug. This is because the *self-destruct-statements* are more common than *function* type variables, *HuangGai* only needs to inject *suicide contracts* bugs by invalidating the security measures of the *self-destruct-statements*.

5 RELATED WORK

Bug injection tools. Some researchers provide bug injection tools to build large-scale vulnerable program datasets. Bonett et al. propose *μSE* [1], a mutation-based Android static analysis tool evaluation framework, which systematically evaluates Android static analysis tools through mutation analysis to find defects in the analysis tools. Their work validates the role of bug injection tools in finding defects in analysis tools. Pewny et al. propose *EvilCoder* [31], a bug injection tool that automatically finds the locations of potentially vulnerable source code, and modifies the source code into the actual vulnerable. *EvilCoder* first uses automated program analysis technologies to find functions that can inject bugs, and then makes attacks possible by inserting statements or invalidating security measures. Our work is inspired by *EvilCoder*. Dolan-Gavitt et al. propose *LAVA* [11], a bug injection tool based on dynamic taint analysis. *LAVA* can quickly inject a large number of bugs into the program 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* [16], 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 use *SolidiFI* to inject 9,369 bugs of 7 types into 50 contracts, and then use these contracts to evaluate 6 analysis tools. The evaluation results show that these tools cause a large number of false positives and false negatives.

Evaluating smart contract analysis tools. Some studies are devoted to evaluating the bug detection abilities of smart contract analysis tools. Zhang et al. [42] propose an Ethereum smart contract bug classification framework, and construct a dataset matching the framework. They use the dataset they constructed to evaluate 9 analysis tools and obtain some interesting findings. Chen et al. [6] evaluate the abilities of 6 analysis tools to identify smart contract control flow transfer, and 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. [12] conduct a large-scale evaluation of 9 analysis tools using 47,587 contracts. They find that these tools would produce a large number of false positives and false negatives. Besides, they present *SmartBugs* [15], an execution framework that integrates 10 analysis tools.

Exploiting smart contract bugs. Some studies' purposes are to find and exploit the bugs in smart contracts. Jiang et al. [19] propose *ContractFuzzer*, a smart contract fuzzing tool. *ContractFuzzer* generates fuzzing inputs and uses these inputs to run smart contracts to try to trigger bugs in smart contracts. Zhang et al. propose *ETPLOIT* [43], a fuzzing-based smart contract bug exploit generator. *ETHPLOIT* uses static taint analysis to generate transaction sequences that can trigger bugs and uses the dynamic seed strategy to overcome hard constraints in the execution paths. Feng et al. propose *SMARTSCOPY* [41], an automatic generation tool for adversarial contracts. *SMARTSCOPY* uses techniques such as symbolic execution to identify and exploit bugs in the victim's smart contracts. Krupp et al. [22] first define the vulnerable contracts, and then they present *TEETHER*, *TEETHER* can generate exploits by analyzing the bytecode of the contracts.

6 CONCLUSION

In this paper, we first introduce an automated approach for injecting 20 types of bugs into Ethereum smart contracts and implement *HuangGai* based on our approach. Secondly, we conduct large-scale experiments to verify that *HuangGai* can inject more types of bugs more accurately than similar tools. Then, we use 6 widely used analysis tools to detect bugs injected by *HuangGai*, and find that these tools cannot detect most of the bugs injected by *HuangGai*, which means that users can use *HuangGai* to find defects in the analysis tools. Finally, we use *HuangGai* to construct 3 datasets, these datasets are expected to become the benchmarks for evaluating analysis tools.

REFERENCES

[1] Richard Bonett, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk. 2018. Discovering Flaws in Security-Focused Static Analysis Tools for Android Using Systematic Mutation. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, USA, 1263âĂŞ1280.

[2] Vitalik Buterin et al. 2013. Ethereum white paper. *GitHub repository* (2013), 22–23.

[3] ChainSecurity. 2020. A securify scanner for Ethereum smart contracts. https://securify.chainsecurity.com/

[4] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiapu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. 2020. GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts. *IEEE Transactions on Emerging Topics in Computing* (2020).

[5] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 442–446.

[6] T. Chen, Z. Li, Y. Zhang, X. Luo, T. Wang, T. Hu, X. Xiao, D. Wang, J. Huang, and X. Zhang. 2019. 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)*. 1–11. https://doi.org/10.1109/ESEM.2019.8870156

[7] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. 2018. Towards saving money in using smart contracts. In *In Proc. IEEE/ACM International Conference on Software Engineering.*

[8] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. TokenScope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in Ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1503–1520.

[9] ConsenSys. 2020. Security analysis tool for EVM bytecode. https://github.com/ConsenSys/mythril

[10] crytic. 2020. Static Analyzer for Solidity. https://github.com/crytic/slither

[11] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. 110–121. https://doi.org/10.1109/SP.2016.15

[12] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz. 2020. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 530–541.

[13] Ethereum. 2020. The development documents of Solidity. Retrieved December 26, 2020 from https://docs.soliditylang.org/en/v0.8.0/

[14] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. 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, 8–15.

[15] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu. 2020. SmartBugs: A Framework to Analyze Solidity Smart Contracts. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1349–1352.

[16] Asem Ghaleb and Karthik Pattabiraman. 2020. 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 (ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 415–427. https://doi.org/10.1145/3395363.3397385

[17] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. EtherTrust: Sound static analysis of ethereum bytecode. *Technische Universität Wien, Tech. Rep* (2018).

[18] GitHub Inc. 2020. Open-source project repository. Retrieved January 4, 2021 from https://github.com/

[19] B. Jiang, Y. Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 259–269. https://doi.org/10.1145/3238147.3238177

[20] S. Jiang, J. Chen, Y. Zhang, J. Qian, R. Wang, and M. Xue. 2018. Evolutionary approach to generating test data for data flow test. *IET Software* 12, 4 (2018), 318–323. https://doi.org/10.1049/iet-sen.2018.5197

[21] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts.. In *NDSS*.

[22] Johannes Krupp and Christian Rossow. 2018. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. 1317–1333.

[23] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 65–68.

[24] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 254–269.

[25] MAIAN-tool. 2020. MAIAN: automatic tool for finding trace vulnerabilities in Ethereum smart contracts. https://github.com/MAIAN-tool/MAIAN

[26] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1186–1189. https://doi.org/10.1109/ASE.2019.00133

[27] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 653–663.

[28] Trail of Bits. 2020. Examples of Solidity security issues. https://github.com/crytic/not-so-smart-contracts

[29] OpenZeppelin. 2020. Web3/Solidity based wargame. https://ethernaut.openzeppelin.com/

[30] Reza M. Parizi, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Amritraj Singh. 2018. 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 (CASCON '18)*. IBM Corp., USA, 103âĂŞ113.

[31] Jannik Pewny and Thorsten Holz. 2016. EvilCoder: Automated Bug Insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC '16)*. Association for Computing Machinery, New York, NY, USA, 214âĂŞ225. https://doi.org/10.1145/2991079.2991103

[32] Scrapy. 2020. Scrapy, a fast high-level web crawling and scraping framework for Python. Retrieved December 26, 2020 from https://github.com/scrapy/scrapy

[33] SmartContractSecurity. 2020. Smart Contract Weakness Classification and Test Cases. https://swcregistry.io/

[34] smartdec. 2020. a static analysis tool that detects vulnerabilities and bugs in Solidity programs. https://tool.smartdec.net/

[35] SMARX. 2020. Warmup. https://capturetheether.com/challenges/

[36] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 9–16.

[37] Christof Ferreira Torres, Julian Schütte, et al. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 664–676.

[38] trailofbits. 2020. Symbolic execution tool. https://github.com/trailofbits/manticore

[39] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 67–82.

[40] Y. Xiaoyong, L. Ying, W. Zhonghai, and L. Tiancheng. 2014. Dependability Analysis on Open Stack IaaS Cloud: Bug Anaysis and Fault Injection. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*. 18–25. https://doi.org/10.1109/CloudCom.2014.10

[41] Rastislav Bodik Yu Feng, Emina Torlak. 2019. Precise Attack Synthesis for Smart Contracts. *arXiv preprint arXiv:1902.06067* (2019).

[42] P. Zhang, F. Xiao, and X. Luo. 2020. A Framework and DataSet for Bugs in Ethereum Smart Contracts. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 139–150. https://doi.org/10.1109/ICSME46990.2020.00023

[43] Q. Zhang, Y. Wang, J. Li, and S. Ma. 2020. EthPloit: From Fuzzing to Efficient Exploit Generation against Smart Contracts. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 116–126. https://doi.org/10.1109/SANER48275.2020.9054822

[44] W. Zou, D. Lo, P. S. Kochhar, X. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu. 2019. Smart Contract Development: Challenges and Opportunities. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2942301