HuangGai: An Ethereum Smart Contract Bug Injection Framework

Anonymous Author(s)

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 issue prevents carrying out reliable performance assessments on 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. *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) it can only inject 7 types of bugs; (2) its injection accuracy is low; (3) it cannot accurately label the locations of the injected bugs. To address the above limitations, we propose a novel approach to enable automatic bug injection for Ethereum smart contracts. Based on this approach, we develop an open-source tool, named *HuangGai*, which can inject 20 types of bugs into smart contracts via analyzing the contracts' control and data flows. The extensive evaluations show that *HuangGai* outperforms *SolidiFI* on both the number of injected bug types and injection accuracy. The experimental results also reveal that the existing analysis tools can only partially detect the bugs injected by *HuangGai*. By means of *HuangGai*, users can generate a large scale of smart contracts with diverse bugs for performing more reliable evaluations of smart contract 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, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Smart contracts are autonomous programs running on blockchain [58]. Ethereum is currently the largest platform that supports smart contracts [5]. 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[17]. 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 smart contract bugs [9, 10, 12, 13, 21, 26, 30, 32, 34, 35, 40, 49, 50, 53]. 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) [19, 41, 42, 45, 47]. Unfortunately, such buggy smart contracts have the following issues:

- *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 example, libraries are widely used to create *real contracts*, whereas they are rarely used to create handwritten contracts. Consequently, such manually constructed contracts cannot be utilized to truly verify the performance of analysis tools on *real contract* bug detection.
- *The code size of contracts is generally small.* By investigating 5 widely used datasets with manually constructed smart contracts (i.e., [19, 41, 42, 45, 47]) and a dataset with *real contracts* (i.e., the dataset consists of 66,205 *real contracts* collected by us), we find that the average number of *loc* (line of code) per manually constructed 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 manually constructed smart contracts usually have a small number of samples (around 100 contracts). When it comes to each type of bugs, the contracts that contain such bugs are much sparser. Therefore, 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. Ghaleb et al. [23] propose the first and only Ethereum smart contract bug injection tool, *SolidiFI*. *SolidiFI* injects bugs by inserting buggy code snippets (i.e., code snippets containing bugs) into all possible locations in a contract. However, *SolidiFI* suffers from the following limitations:

ISSTA 2021, 12-16 July, 2021, Aarhus, Denmark

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- *Most of the bugs injected by SolidiFI are easy to detect. SolidiFI* creates bugs by directly inserting pre-made buggy code snippets into contracts without using any existing structure of the contracts. The code snippets inserted by *SolidiFI* are usually simple and independent of the contract's original control and data flows (eg., the code snippet shown in Fig 1), which enable analysis tools to capture the injected bugs without an in-depth analysis of the contract.
- *SolidiFI can only inject 7 types of bugs.* Existing studies [55] show that there are far more than 7 types of Ethereum smart contract bugs.
- *SolidiFI cannot accurately inject bugs.* Certain bugs injected by *SolidiFI* cannot be exploited by any external attacker. For instance, in a code snippet inserted by *SolidiFI* like Fig 2, *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).
- *SolidiFI cannot label bug locations precisely.* A typical labelling error of *SolidiFI* is illustrated in Fig 3. *SolidiFI* labels the code on line 10 contributing to a *re-entrancy* bug. Nevertheless, line 10 does not contain any statement.

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

Figure 1: A code snippet inserted by *SolidiFI* that contains a *wasteful contracts* bug

```
1  mapping(address => uint) redeemableEther_re_ent11;  //SolidiFI label.
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)("");  //Right label.
7      redeemableEther_re_ent11[msg.sender] = 0;
8  }
```

Figure 2: A bug that *SolidiFI* cannot inject accurately

```
10  }
11
12  event Transfer(address indexed _from, address indexed _to, uint _value);
```

Figure 3: *SolidiFI*'s label error

To address the above limitations, we propose *HuangGai*[1], a novel Ethereum smart contract bug injection tool. *HuangGai* extracts data flows and control flows from the collected *real contracts*. Next, *HuangGai* ascertains the contracts suitable for bug injection by analyzing these contracts' data flows and control flows. Simultaneously, *HuangGai* identifies the proper bug injection locations and bug statements from these contracts. Eventually, up to 20 types of bugs can be injected into the smart contracts by *HuangGai* .

Our primary contribution is fourfold:

(1) We design a novel approach to automatically inject bugs into Ethereum smart contracts. Based on this approach, we implement an open-source tool, *HuangGai*, which can inject up to 20 types of bugs into contracts. It is worth noting that, among these 20 types of bugs, there are 4 types that have not been covered by any current analysis tool. Buggy contracts containing these 4 types of bugs are expected to assist researchers to further enhance the detection capability of the analysis tools.

(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 label bug locations more precisely, compared to *SolidiFI*.

(3) We employ a group of state-of-the-art analysis tools [6, 14, 15, 36, 46, 52] to detect the buggy contracts generated by *HuangGai*. The detection results show that these analysis tools cannot effectively detect most of the bugs injected by *HuangGai*, under the premise that all these tools use the same bug labeling criteria. This indicates that users can use *HuangGai* to find more defects 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 currently the largest *real contract* based buggy contract dataset in terms of the number of verified contracts.
   - *Dataset 2*: This dataset comprises 4,744 non-verified real buggy contracts covering 20 types of bugs. 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 66,205 *real contracts* without injected bugs. Researchers can analyze the contracts in this dataset to gain insights of current Ethereum smart contracts.

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

2 BACKGROUND

2.1 Ethereum smart contract

Ethereum [5] provides a variety of programming languages for developers to create smart contracts. Users deploy the contracts into Ethereum by sending transactions. In Ethereum, each contract or user is assigned a unique address as their identifiers. Ether is the cryptocurrency used by Ethereum. Both contracts and users can trade ethers. To avoid abusing Ethereum's computational resources,

[1]https://anonymous.4open.science/r/4f1fea66-dcec-4bd0-86cd-abc060ff16cc

[2]Users can access these three datasets by visiting https://anonymous.4open.science/r/4f1fea66-dcec-4bd0-86cd-abc060ff16cc/

Ethereum charges fees (called *gas*) for each executed contract statement.

2.2 Solidity

Solidity [20] is currently the most mature and widely used Ethereum smart contract programming language. Solidity is a Turing-complete language and able to express complex logic. 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 corresponding compiler to compile the contract. Solidity assigns a *function selector* to each function. The overridden function has the same *function selector* value as the overriding function. Solidity supports (multiple) inheritance including polymorphism. Solidity 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 terminate the program execution.

2.3 Smart contract bug detection criteria

A number of smart contract bug classification frameworks and corresponding detection criteria have been proposed recently [8, 16, 49, 55]. Among them, Zhang et al. [55] propose a comprehensive smart contract bug classification framework by extending the *IEEE Standard Classification for Software Anomalies* [27], which summarizes 49 types of bugs and their severity levels. Besides, they also propose a set of characteristic-based 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 the injection methods of the 20 most severe bug types, such as *re-entrancy* and *integer overflow and underflow*. Table 2 shows the name of each bug type. The specific description and bug detection criteria of each bug type can be found in [55].

2.4 Smart contract analysis tools

Smart contract analysis tools are designed to perform automatic bug detection. 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 numbers or line number ranges). At present, a variety of techniques have been applied to implement the bug detection functions, such as pattern matching [21, 49], symbolic execution [32, 35, 50], and fuzzing [30].

3 *HUANGGAI*

3.1 Overview of *HuangGai*

HuangGai is an Ethereum smart contract bug injection tool. It can inject up to 20 types of severe bugs into *real contract* source code. These generated bugs can be detected by either bytecode or source code based analysis tools. Based on [55], *HuangGai* employs the characteristic criteria to create 20 types of bugs. The workflow of *HuangGai* is shown in Fig 4. *HuangGai* first collects *real contracts* (performed by *ContractSpider* introduced in Section 3.2). It then ascertains whether a contract is suitable for injecting a certain type of bugs by analyzing the contract's control and data flows and extract the data required for injecting bugs (achieved by *ContractExtractor* depicted in Section 3.4). According to the data required for injecting bugs extracted by *ContractExtractor*, *HuangGai* injects bugs into the contract (implemented by *BugInjector* described in Section 3.5).

3.2 *ContractSpider*

One of our goals is to allow users to use *HuangGai* to automatically create buggy *real contracts* without manually collecting the contracts beforehand. Accordingly, the first step of *HuangGai* is to collect *real contracts* as the sources for bug injection. We implement *ContractSpider* (Fig 4) based on the work of [44], which is a parallel high-performance web crawler. *ContractSpider* automatically collects contract source code by crawling open-source *real contracts* websites (eg., http://etherscan.io/), followed by saving these source code as Solidity files. Note that users only need to run *ContractSpider* once to collect all the *real contracts*.

3.3 Construct a contract's control and data flows

HuangGai needs to analyze the control and data flows of a contract to ascertain 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* utilizes the same method to construct contracts' data flows and control flows.

Specifically, *HuangGai* needs to construct all the function-call paths in a contract (even if the contract is a derivative contract) to track the definitions and use of data in each path. Based on *solc*[3] and *Slither*[4], *HuangGai* is able to construct a contract's control and data flows:

- **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: *definition-use* pairs [31] of data and linear inheritance orders. Based on the above information, *HuangGai* can track the definition and use of all the data.
- **Control flow**: Based on the control flow graph (*CFG*) of each function and the 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 function modifier) overrides the same function (or function modifier) of the base contract, *HuangGai* uses the *function selector* to identify the overridden function in the path and replaces it with the overriding function (algorithm 1 shows the specific process).

3.4 *ContractExtractor*

Our another goal is to inject bugs into a contract while modifying the contract (source code) as little as possible. It requires *HuangGai* to ascertain if a contract is qualified for being injected with a certain

[3]Solidity official compiler, https://github.com/ethereum/solc-js

[4]A widely used *Solidity* static analysis framework, https://github.com/crytic/slither

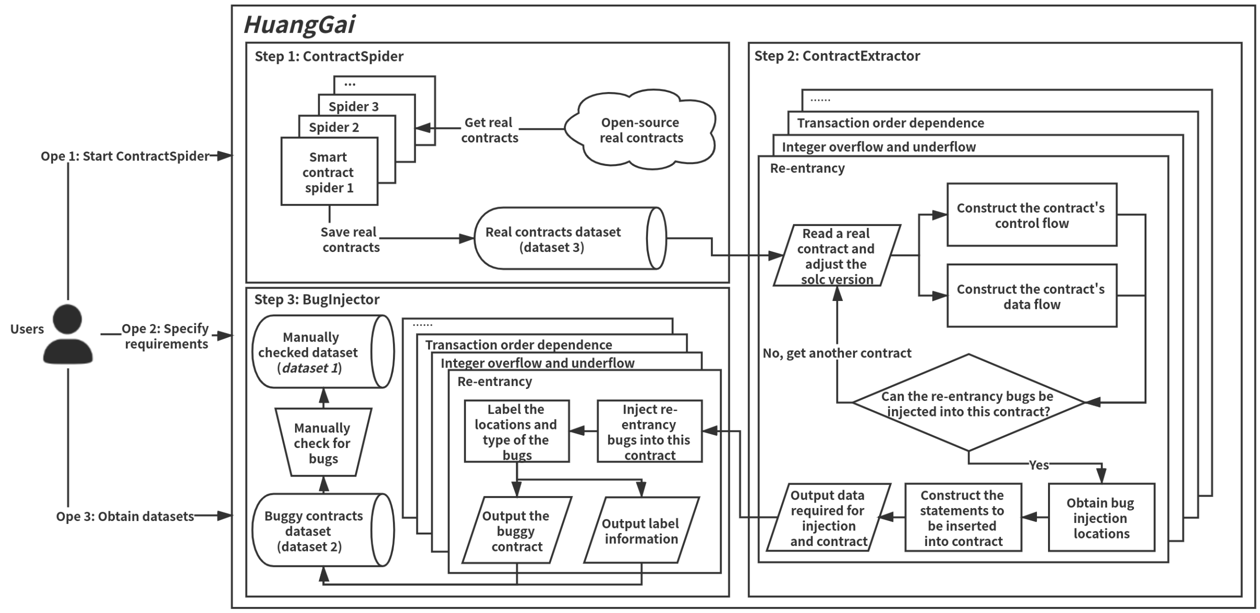

Figure 4: *HuangGai* workflow

type of bugs. For instance, if *HuangGai* wants to inject *integer overflow and underflow* bugs into a contract that does not contain integer arithmetic statements, then *HuangGai* needs to perform a series of operations (eg., declaring integer variables, inserting arithmetic statements, etc.) to inject *integer overflow and underflow* bugs, which may damage the authenticity and logic consistency of the injected bugs.

ContractExtractor is designed to achieve this goal. *ContractExtractor* reads a contract collected by *ContractSpider*, constructs the contract's control and data flows, and analyzes the data and control flows to ascertain 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, then *ContractExtractor* will pass the contract and the data required for bug injection to *BugInjector*; otherwise *ContractExtractor* will skip this contract. It is noteworthy that the internal workflows and employed techniques of *ContractExtractor* for different types of bugs are distinct. Due to space limitations, in this paper, we select the *ContractExtractor* for two representative bugs, *re-entrancy* and *integer overflow and underflow*, as examples to describe how *ContractExtractor* works[5].

3.4.1 ContractExtractor for re-entrancy (CER). According to [55], the characteristic of a *re-entrancy* bug is that a contract uses *call-statements* to deposit ethers into a contract and then repeatedly controls the *call-statements* to deduct the number of tokens held by the payee address. This would trigger multiple invocations of the *call-statements* and even drain the deposit contract. Since *HuangGai*

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

can employ the *call-statements* to send ethers into a contract, 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 as *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 as *ledgers*).

CER searches for the *ledgers* and *deduct-statements* of a contract 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 meeting the following conditions: 1) The entry function of this path is a function declared as ***payable***. 2) There is at least one value increment operation on a *mapping(address=>uint256)*

[5]For *ContractExtractor*(s) of the remaining 18 types of bugs, please visit https://anonymous.4open.science/r/4f1fea66-dcec-4bd0-86cd-abc060ff16cc/18 types of bugs injection methods.pdf

Algorithm 1: Contract CFG construction algorithm

```
Input: Contract's source code SC
Output: ledgerSet, deductSet
changeSolcVersionBySolc(SC);
// Adjust the local solc version to compile the
   contract.
ContractAstSet C = getContractASTBySolc(SC);
Set P;
// function-call paths set.
foreach contractAst in C do
    // from base contract to derived contract
    callGraphSet = getFuncCallGraphBySlither(SC);
    CFGSet = getFuncCFGBySlither(SC);
    contractPathSet =
     getContractCFG(callGraphSet, CFGSet);
    funcAndItsSelector =
     getFuncSelectorBySolc(contractAst);
    P = P ∪ contractPathSet;
    // Add new function-call paths.
    foreach path in P do
        foreach (oldFunc, oldSelector) in path do
            foreach (newFunc, newSelector) in
             funcAndItsSelector do
                if newSelector == oldSelector then
                    /* Override */
                    oldFunc is replaced by newFunc;
                end
            end
        end
    end
end
return P;
```

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*.

(2) **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*.

(3) **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*. When 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 bug injection.

3.4.2 *ContractExtractor for integer overflow and underfow (CEI).*

According to [55], 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 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 }
```

Figure 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., detecting whether the *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* will be able to 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 passed by its corresponding *ContractExtractor* and performs bug injection and labeling. According to the data passed by *ContractExtractor*, 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.
- Invalidating security measures. *BugInjector* first invalidates the security measures in the contract, followed by labeling the statements without security protection as bugs.

Due to space limitations, we use the *BugInjectors* of *re-entrancy* and *integer overflow and underflow* as examples to describe how *BugInjector* works. These two *BugInjectors* use the aforementioned means to inject bugs[6].

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

Figure 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 *re-entrancy* bugs in the lines where the *call-statements* are inserted.

[6]For *BugInjector*(s) of the remaining 18 types of bugs, please visit https://anonymous.4open.science/r/4f1fea66-dcec-4bd0-86cd-abc060ff16cc/18 types of bugs injection methods.pdf

3.5.2 *BugInjector of integer overflow and underfow (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* unprotected. Next, it labels *IOA* bugs behind the unprotected statements in the *target*.

4 EVALUATION

Compared with state-of-the-art tools, we conduct a series of experiments to respond to the following research questions:

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

RQ1 tries to compare the performance between *HuangGai* and existing bug injection tools on successfully injecting activated bugs (i.e., a bug that can be activated means that external attackers can exploit the bug). RQ2 aims to assess how bugs injected by these tools challenge existing smart contract analysis tools. RQ3 intends to measure the bug injection speed of these tools.

4.1 Tools used in the evaluation

We employ two types of bug injection 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 use these analysis tools to assess the bug injection performance of *HuangGai* and *SolidiFI*.

The state-of-the-art smart contract analysis tools are collected via the following two channels:

- Analysis tools that have been covered by the latest empirical review papers, i.e., [19, 23].
- Analysis tools that are available on GitHub [29]. We use the keywords *smart contract security* and *smart contract analysis tools* to search in Github [29]. We select the top twenty tools (sorted by the number of *stars* in a descending order) 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 can support command-line interface. This allows us to automatically run analysis tools to detect bugs.
- *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 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 selection criteria, we finally select 6 analysis tools to detect bugs injected by *HuangGai* and *SolidiFI*. Table 1 lists the 6 selected tools and the excluded tools.

Table 1: Selected and excluded tools based on our selection criteria

	Selection criteria	Tools that violate the criteria
Excluded	criterion 1	*teEther [33], Zeus [32], ReGuard [34], SASC [56], Remix, sCompile [7], Ether, Gasper [10]*
	criterion 2	*Vandal [4], Echidna [25], MadMax [24], VeriSol [54]*
	criterion 3	*EthIR [1], E-EVM, Erays [57], Ethersplay, EtherTrust [26], contractLarva [2], FSolidM [37], KEVM, SolMet, Solhint, rattle, Solgraph, Octopus, Porosity [48]*
	criterion 4	*Osiris [50], Oyente [35], HoneyBadger [51]*
Selected	*Maian [40], Manticore [38], Mythril [39], Securify [53], Slither [21], SmartCheck [49]*	

4.2 Dataset for evaluation and environment

We first employ *HuangGai* and *SolidiFI* to generate an *injected contract* (i.e., contracts with injected bugs) dataset. The original *real contracts* are sourced from *dataset 3* depicted 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 evaluation datasets. Specifically, for each type of bugs, we first use *HuangGai* to generate 50 *injected contracts*. We then utilize *SolidiFI* to inject the same bugs into the same group of contracts. Finally, we obtain a dataset comprising 964 *injected contracts* (covering 20 types of bugs) generated by *HuangGai* and 323 *injected contracts* (covering 7 types of bugs) generated by *SolidiFI*. This is because *HuangGai* and *SolidiFI* cannot find 50 qualified contracts from the dataset for certain types of bugs.

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

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

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 external attackers and cause the expected consequences. Three smart contract debugging experts manually check each injected bug and reach a consensus through discussions. If an injected bug cannot be exploited by external attackers, the bug will not be deemed to be activated.

A common issue of *SolidiFI* is that it sometimes claims a bug injection. However, no bugs have been actually injected. To address this issue, we manually check each of claimed injected bugs and only count the actually injected bugs.

Results. Table 2 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. It can be seen 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. The *accuracyRate(s)* of *HuangGai* reach 100% for 13 types of bugs.

Analysis. The main reasons for the injection failures of *SolidiFI* are: 1) *SolidiFI* is incompatible with Solidity 0.6.0 and subsequent versions. This leads to a large number of compilation errors. Specifically, there are a lot of syntax errors in the *injected contracts* when using *SolidiFI* to inject bugs into the contracts of Solidity 0.6.0 and subsequent versions, due to the grammatical changes caused by Solidity version upgrade. 2) There are problems with the implementation of *SolidiFI*. When *SolidiFI* performs *results of contract execution affected by miners* bug injection, *SolidiFI* inserts a piece of text into the contracts. This 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* can cause dead code of these bugs. 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 contracts, the statements inserted by *HuangGai* may still cause compilation errors. However, these errors are not common. Only two compilation errors occur in the process of compiling all the 964 *injected contracts* generated by *HuangGai*. 2) Excessive security measures. In some contracts, developers use multiple mutually redundant security measures to prevent bugs. This generates multiple security measures in the contracts to prevent the injected bugs from being activated, though *HuangGai* has invalidated most of the security measures.

4.4 RQ2: Analysis tool based evaluation

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

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

where *BDN* represents the number of bugs detected by a analysis tool, and *BIN* represents the number of bugs injected by a bug injection tool. The lower the *captureRate*, the more difficult it is for

Table 2: The *accuracyRate(s)* and *speed(s)* of *HuangGai(HG)* and *SolidiFI(SF)* for bug injection

	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

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 15 mins per contract and bug type.

Results. *SolidiFI* cannot provide the exact locations (i.e., line numbers) of injected bugs. Instead, it 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, 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* (eg., as shown in Fig 2). 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*. 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 3 shows the *captureRate*(s) of *HuangGai* (based on the detection criterion of *line matching*) and *SolidiFI* (based on the detection criterion of *range matching*). *N/A* represents that a bug injection tool is not designed to inject a type of bugs. * represents that an analysis tool is not designed to detect a type of bug. # represents that a type of bugs injected by a bug injection tool cannot be activated. It can be seen from Table 3 that the *captureRate*(s) of *SolidiFI* is still **very low** even based on the detection criterion of *range matching*. We randomly select 162 (accounting for 50% of the contracts generated by *SolidiFI*) *injected 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 the *loc* attributes of the bugs labelled by *SolidiFI* are **mostly incorrect** (eg., as shown in Fig 3), which leads to the low *captureRate*(s) of *SolidiFI*. In contrast, the *captureRate*(s) of *HuangGai* is **much higher** than that of *SolidiFI* for each analysis tool on most comparable bug types. This demonstrates the higher bug labelling precision of the former. In addition, the following facts attract our attention: *SolidiFI* respectively injects 1,390 *use* tx.origin *for authentication* bugs and 1,228 *wasteful contracts* bugs into 50 contracts, and *Slither* reports respectively 1,390 and 1,244 of the bugs from these contracts. It is not uncommon that the numbers of bugs for a certain type reported by the analysis tools are **close to** what are injected by *SolidiFI*. This indicates that the analysis tools have little room to improve with *SolidiFI*. In contrast, it can be seen from Table 3 that these analysis tools cannot detect most of the bugs injected by *HuangGai* with sound *captureRate*(s). This implies that the existing analysis tools may have large room for both functional and performance improvement.

Analysis. The non-symbolic-execution tools (i.e., *SmartCheck* and *Slither*) show **better** detection performance when detecting bugs injected by *HuangGai*. This is because these tools usually use pattern matching to detect bugs and consequently they are able to detect a large number of bugs. Meanwhile they also generate many **false positives**. For instance, although *Slither* can detect all the 24 *re-entrancy* bugs injected by *HuangGai*, in order to capture these *re-entrancy bugs*, *Slither* reports a total of 387 *re-entrancy warnings*. The symbolic-execution tools show **weaker** bug detection capability. This is because these tools exceed the timeout value when

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%	2.2%	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%	*	*	*	*
	SF	7.6%	0.0%	*	*	*	*
Re-entrancy	HG	*	100.0%	16.7%	0.0%	*	26.1%
	SF	#	#	#	#	*	#
Wasteful contracts	HG	*	62.6%	0.7%	*	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%	12.2%	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	*	*	0.0%	*	*	*
	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 variablesare accessed by *public*/*external* function	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%	*	*	*	*
	SF	N/A	N/A	N/A	N/A	N/A	N/A

analyzing many contracts. Although we set a long timeout value (15 mins), symbolic-execution tools need to take much longer time to cover all the paths of the contracts due to the high complexity of the *injected contracts*. As a whole, the existing analysis tools can detect part bugs (12 of 20) injected by *HuangGai* with **relatively higher** accuracy (i.e. >60%). This indicates that *HuangGai* has reached a consensus with the developers of the analysis tools in terms of a majority of bug labeling criteria. 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.

4.5 RQ3: Bug injection efficiency

We measure the injection time of *HuangGai* and *SolidiFI* when constructing the evaluation dataset. The following formula is utilized to calculate the bug injection speed of *HuangGai* and *SolidiFI*:

$$speed = IT \div BIN \tag{3}$$

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*.

Results. Table 2 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*.

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 for constructing the contract's control and data flows 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 8. 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.

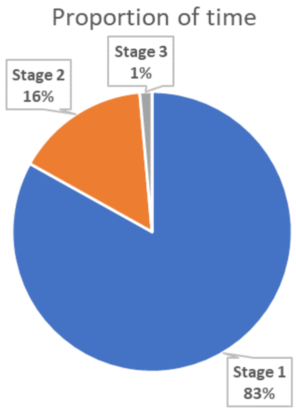

Figure 8: The proportion of the time for each stage of the bug injection process

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 210.6 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 RELATED WORK

Bug injection tools. Some researchers develop bug injection tools to build large-scale vulnerable program datasets. Bonett et al. propose *μSE* [3], a mutation-based Android static analysis tool evaluation framework. It systematically evaluates Android static analysis tools through mutation analysis to detect defects of these tools. Their work validates the role of bug injection tools in finding defects in analysis tools. Pewny et al. propose *EvilCoder* [43], a bug injection tool that automatically finds the locations of potentially vulnerable source code. It modifies the source code and outputs the actual vulnerable. *EvilCoder* first employs automated program analysis technologies to find functions for bug injection. Further, it conducts possible attacks by inserting statements or invalidating security measures. Our work is inspired by *EvilCoder*. Dolan-Gavitt et al. propose *LAVA* [18], 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* [23], 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. These bug injection tools cannot inject bugs into Ethereum smart contracts except for *SolidiFI*.

Evaluating smart contract analysis tools. Some studies are devoted to evaluating the bug detection performance of smart contract analysis tools. Zhang et al. [55] 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. [11] 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. [19] 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* [22], an execution framework that integrates 10 analysis tools. 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.

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* [45] provides a list of 36 types of Ethereum smart contract bugs and creates the exemplary buggy contracts for each type of bug in the list. *Crytic* [41] 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. [55] propose an Ethereum smart contract bug classification framework that covers the currently highest number (49) of bug types. They also provide a buggy contract dataset to exemplify the bug types in the classification framework. This dataset is currently the largest handwritten dataset in terms of the number (173) of contracts. Durieux et al. [19] 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* [28], 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 CONCLUSION

In this paper, we introduce an approach to automatically inject 20 types of bugs into Ethereum smart contracts. We implement a bug injection tool, *HuangGai*, based on this approach. Next, we conduct large-scale experiments to verify that *HuangGai* can inject more types of bugs than state-of-the-art tools with higher accuracy. We select 6 widely used analysis tools to detect the bugs injected by *HuangGai*. The experimental results demonstrate that *HuangGai* can better indicate the defects of these tools. Finally, we use *HuangGai* to construct 3 *real contract* datasets. These datasets are expected to be utilized as the benchmarks for evaluating all Ethereum smart contract bug analysis tools.

REFERENCES

[1] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. 2018. Ethir: A framework for high-level analysis of ethereum bytecode. In *International symposium on automated technology for verification and analysis*. Springer, 513–520.

[2] Shaun Azzopardi, Joshua Ellul, and Gordon J Pace. 2018. Monitoring smart contracts: Contractlarva and open challenges beyond. In *International Conference on Runtime Verification*. Springer, 113–137.

[3] 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, 1263–1280.

[4] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981* (2018).

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

[6] ChainSecurity. 2020. A SECURITY SCANNER FOR ETHEREUM SMART CONTRACTS. https://securify.chainsecurity.com/

[7] Jialiang CHANG, Bo GAO, Hao XIAO, Jun SUN, Yan CAI, and Zijiang YANG. 2019. sCompile: Critical path identification and analysis for smart contracts.(2019). In *International Conference on Formal Engineering Methods (ICFEM 2019), Shenzhen, China*. 5–9.

[8] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen. 2020. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering* (2020), 1–1.

[9] 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).

[10] 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.

[11] 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

[12] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. 2018. 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, 81–84.

[13] 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.

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

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

[16] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. 2016. 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, 79–94.

[17] ConsenSys Diligence. [n. d.]. Ethereum Smart Contract Security Best Practices. https://consensys.github.io/smart-contract-best-practices/

[18] 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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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

[24] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.

[25] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 557–560.

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

[27] ISDW Group et al. 2010. 1044-2009-IEEE Standard Classification for Software Anomalies. *IEEE, New York* (2010).

[28] NCC Group. [n. d.]. Decentralized Application Security Project. https://dasp.co/

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

[30] Bo Jiang, Ye Liu, and WK Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 259–269.

[31] 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

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

[33] 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.

[34] 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.

[35] 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.

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

[37] Anastasia Mavridou and Aron Laszka. 2018. Tool demonstration: FSolidM for designing secure Ethereum smart contracts. In *International conference on principles of security and trust*. Springer, 270–277.

[38] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem 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)*. IEEE, 1186–1189.

[39] Bernhard Mueller. 2018. Smashing ethereum smart contracts for fun and real profit. In *9th Annual HITB Security Conference (HITBSecConf)*. 54.

[40] 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.

[41] Trail of Bits. 2021. Examples of Solidity security issues. https://github.com/crytic/not-so-smart-contracts

[42] OpenZeppelin. 2021. Web3/Solidity based wargame. https://ethernaut.openzeppelin.com/

[43] 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, 214–225. https://doi.org/10.1145/2991079.2991103

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

[45] SmartContractSecurity. 2021. Smart Contract Weakness Classification and Test Cases. https://swcregistry.io/

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

[47] SMARX. 2021. Warmup. https://capturetheether.com/challenges/

[48] Matt Suiche. 2017. Porosity: A decompiler for blockchain-based smart contracts bytecode. *DEF con* 25, 11 (2017).

[49] 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.

[50] 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.

[51] Christof Ferreira Torres, Mathis Steichen, et al. 2019. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1591–1607.

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

[53] 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.

[54] Yuepeng Wang, Shuvendu K Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. 2019. Formal verification of workflow policies for smart contracts in azure blockchain. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 87–106.

[55] 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

[56] Ence Zhou, Song Hua, Bingfeng Pi, Jun Sun, Yashihide Nomura, Kazuhiro Yamashita, and Hidetoshi Kurihara. 2018. Security assurance for smart contract. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 1–5.

[57] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. 2018. Erays: reverse engineering ethereum's opaque smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1371–1385.

[58] 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