SCAnoGenerator: Automatic Anomaly Injection for Ethereum Smart Contracts

Pengcheng Zhang, Ben Wang, Xiapu Luo, and Hai Dong

Abstract—Although many tools have been developed to detect anomalies in smart contracts, the evaluation of these analysis tools has been hindered by the lack of adequate anomalistic *real-world contracts* (i.e., smart contracts with addresses on Ethereum to achieve certain purposes). This problem prevents conducting reliable performance assessments on the analysis tools. An effective way to solve this problem is to inject anomalies into *real-world contracts* and automatically label the locations and types of the injected anomalies. *SolidiFI*, as the first and only tool in this area, was developed to automatically inject anomalies into Ethereum smart contracts. However, *SolidiFI* is subject to the limitations from its methodologies (e.g., its injection accuracy and authenticity are low). To address these limitations, we propose an approach called *SCAnoGenerator*. *SCAnoGenerator* supports Solidity 0.5.x, 0.6.x, 0.7.x and enables automatic anomaly injection for Ethereum smart contracts via analyzing the contracts' control and data flows. Based on this approach, we develop an open-source tool, which can inject 20 types of anomalies into smart contracts. The extensive experiments show that *SCAnoGenerator* outperforms *SolidiFI* on the number of injected anomaly types, injection accuracy, and injection authenticity. The experimental results also reveal that existing analysis tools can only partially detect the anomalies injected by *SCAnoGenerator*.

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

1 Introduction

Code anomalies are widely present and inevitable in traditional software. Code anomalies usually refer to abnormal behaviors of software code, which greatly restrict the lifespan of software projects [1]. When an anomaly occurs in the code, the program may crash, generate error messages, or perform incorrect operations. Smart contracts are autonomous programs running on blockchain [2]. Ethereum is currently the largest platform that supports smart contracts [3]. Lots of applications based on smart contracts have been developed and deployed on Ethereum. Similar to traditional computer programs, it is difficult to avoid anomalies[1] in smart contracts. Code anomalies in smart contracts may be caused by logical errors, memory overflows, resource leaks, and other issues [4]. Recent years have witnessed many attacks that exploit the anomalies in smart contracts to cause severe financial loss [5]. Even worse, the deployed contracts cannot be modified for anomaly patching, while a logic contract can sometimes be replaced by a proxy contract at runtime. Hence, it is essential to detect and fix all anomalies in the contracts before deployment.

Recent studies have developed many tools for detecting anomalies in smart contract [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26]. 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 anomalies. Note that existing evaluations mainly rely on *handwritten contracts* (i.e., contracts that are manually constructed by researchers according to the characteristics of known anomalies) [27], [28], [29], [30], [31]. Unfortunately, such anomalistic smart contracts have the following problems:

- *Lack of real business logic*. Since these manually constructed contracts are only used for assessing the performance of anomaly detection, they are usually different from *real-world contracts* in terms of the statement types, control structure types, and programming patterns, etc. For instance, libraries are widely used to create *real-world contracts*, whereas they are rarely used in *handwritten contracts*. Consequently, such *handwritten contracts* cannot be utilized to truly verify the performance of analysis tools on detecting anomalies in *real-world contracts*.
- *The code size of contracts is generally small*. By investigating 5 widely used datasets with *handwritten contracts* (i.e., [27], [28], [29], [30], [31]) and a dataset with 66,208 *real-world contracts* collected by us, we find that the average number of *loc* (line of code) per *handwritten contract* is 42, in comparison to 376 *loc* per *real-world 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-world contracts*.
- *The number of contracts with diverse anomalies is inadequate*. The datasets with *handwritten contracts* usually have a small number of samples (around 100 contracts). When it comes to each type of anomalies,

- *P. Zhang, and B. Wang are with the Key Laboratory of Water Big Data Technology of Ministry of Water Resources, Hohai University & the College of Computer Science and Software Engineering, Hohai University, Nanjing, China E-mail: pchzhang@hhu.edu.cn; shawnwb@163.com*
- *X. Luo is with Department of Computing, the Hong Kong Polytechnic E-mail: csxluo@comp.polyu.edu.hk*
- *H. Dong is with the School of Computing Technologies, RMIT University, Melbourne, Australia E-mail: hai.dong@rmit.edu.au*

Manuscript received XXX, XXXX; revised XXX, XXXX.(Corresponding authors: Pengcheng Zhang; Ben Wang.)

1. We use broader term anomalies to encompass any unexpected behaviors of software code instead of bugs or vulnerabilities. Further explanation is provided in Section 2.2.

the contracts that contain certain anomalies are much sparser. Hence, the experimental results upon such a small number of contracts with insufficient and unbalanced types of anomalies would be biased.

The above problems greatly challenge users to find out the true performance of analysis tools on *real-world contract* anomaly detection. An effective way to address the above problems is to inject anomalies into the *real-world contracts* and automatically label the locations and types of the injected anomalies. Researchers have made efforts in this area [32], [33], among which Ghaleb et al. [33] proposed the first and only Ethereum smart contract bug/anomaly injection tool, *SolidiFI*. It uses three ways (i.e., *insert full anomalistic code snippets*, *code transformation*, *weakening security mechanisms*) to inject anomalies into all possible locations in a contract. By investigating *SolidiFI*, we found that the *SolidiFI* methodology has the following three limitations:

1) *The anomalies injected by SolidiFI lacks authenticity.* *SolidiFI* claims that it uses three ways to inject anomalies into the contracts. However, according to our investigation (we used *SolidiFI* to inject 10,627 anomalies into 323 randomly selected *real-world contracts*), most of the anomalies are injected by inserting pre-made full anomalistic code snippets into the contracts, without taking into account the contracts' original structures (the *structures* we refer to here are the structures such as variables or parameters in the contracts).
2) *SolidiFI can only inject a limited number of anomalies.* Most of the anomalies injected by *SolidiFI* are duplicates of pre-made code snippets. For example, our experiments (Section 4.4) found that, although *SolidiFI* claimed that it injected 1,737 *integer overflow and underflow* anomalies into 50 contracts, in fact these anomalies are just duplicates of 40 pre-made code snippets.
3) *SolidiFI may fail to accurately inject anomalies*. Certain anomalies injected by *SolidiFI* cannot be exploited by any attacker. For instance, in a code snippet inserted by *SolidiFI* like Listing 1, *SolidiFI* does not insert any statement into the contract to modify the value of the variable *redeemableEther_re_ent11* (declared in line 1). This makes *redeemableEther_re_ent11[msg.sender]* keep the initial value (0) unchanged and the *require-statement* (line 4) always throws an exception. It eventually invalidates the injected anomaly (line 6).

To address the above limitations, we propose *SCAnoGenerator*[2], an anomaly injection approach for Ethereum smart contracts. *SCAnoGenerator* first collects *real-world contracts* and extracts control and data flows from the *real-world contracts*. Then, it checks whether the contracts are suitable for anomaly injection by analyzing their control and data flows. If that is the case, *SCAnoGenerator* identifies the proper locations for anomaly injection in these contracts and constructs the statements used to inject different types of anomalies. Eventually, up to 20 types of anomalies can be injected into the contracts by *SCAnoGenerator*.

```
1  mapping(address=>uint) redeemableEther_re_ent11;//SolidiFI
       label: loc-1,length-8
2  function claimReward_re_ent11() public{
3  //ensure there is a reward to give
4  require(redeemableEther re ent11[msg.sender]>0);
5  uint transfervalue_re_ent11 = redeemableEther_re_ent11[msg
       .sender];
6  msg.sender.call.value(transfervalue_re_ent11)("");//this
       line causes the anomaly
7  redeemableEther_re_ent11[msg.sender]=0;
8  }
9
10 event Transfer(address indexed _from,address indexed _to,
       uint value);
```

Listing. 1:An example of incapability of *SolidiFI* to accurately inject and precisely label anomalies, where the variable *redeemableEther_re_ent11* (line 1) is simply defined and cannot be modified in the next statements, which leads to the inaccuracy of the inserted anomaly (line 6).

```
1  pragma solidity 0.5.1;
2
3  contract sGcanInjectThisAno_Base{
4  uint256 public number = 0;
5  function overflowHere(uint256 _amount) internal{
6  uint256 tempNumber = number + _amount;
7  //SCAnoGenerator changes the following statement to a
       comment.
8  //require(tempNumber >= number && tempNumber >= _amount);
9  number = tempNumber;//So SCAnoGenerator injects this
       anomaly.
10 }
11 }
12
13 contract sGcanInjectThisAno is sGcanInjectThisAno_Base{
14 function exploit(uint256 _amount) external{
15 overflowHere(_amount);
16 }
17 }
```

Listing. 2:An *integer overflow and underflow* anomaly injected by *SCAnoGenerator*, turning the *require* statement on line 8 into a comment, which invalidates the function *overflowHere* over the variable *tempNumber*.

Our primary contributions are fourfold:

1) We propose an approach called *SCAnoGenerator* to automatically inject anomalies into Ethereum smart contracts. Based on this approach, we implement an open-source tool which can inject up to 20 types of anomalies into contracts. Given a contract, by analyzing the contract's control and data flows, *SCAnoGenerator* can ascertain which types of anomalies are suitable to be injected into the contract. In other words, this functionality empowers *SCAnoGenerator* to inject anomalies by slightly modifying the contract. Besides, *SCAnoGenerator* leverages the structures (e.g., variables, functions, basic structures, etc.) of the original contracts to construct the statements to be inserted into the contract and identifies the proper anomalistic statement injection locations (this makes the locations and style of the anomalies injected by *SCAnoGenerator* vary with the contracts). Hereby, *SCAnoGenerator* can inject more logical and hard-to-find anomalies into the contracts (e.g., the *integer overflow and underflow* anomaly shown in Listing 2).
2) We conduct extensive experiments to evaluate *SCAnoGenerator*. The experimental results show that

2. *SCAnoGenerator* means smart contract anomaly generator, https://github.com/Our4514444/SCAnoGenerator

SCAnoGenerator can effectively inject more types of anomalies with higher accuracy and authenticity, compared to *SolidiFI*. For instance, *SCAnoGenerator* shows equal or higher anomaly injection accuracy than *SolidiFI* for all the 7 comparable types of anomalies.

3) We evaluate a group of state-of-the-art analysis tools [34], [35], [36], [37], [38], [39] by using the anomalistic contracts generated by *SCAnoGenerator*. The detection results show that these analysis tools can only detect 37.3% anomalies injected by *SCAnoGenerator*. In contrast, these detection tools only missed 49.4% of anomalies injected by *SolidiFI*. This validates that *SCAnoGenerator* can be used to uncover more weaknesses in analysis tools.
4) By means of *SCAnoGenerator*, we generate and release the following 3 public datasets[3].Table 3 in Section 4 shows the basic statistics of the contracts (i.e., average and media size (LoC), numbers of functions, function modifiers and received transactions per contract [40]. We also employ box-plots to describe the distribution of received transactions per contract among the datasets in Fig 2.

   - *Dataset 1*: *dataset 1* is a set of simulated anomalistic contracts which has been manually evaluated by human experts. This dataset consists of 964 anomalistic contracts covering 20 types of anomalies. These anomalistic contracts have been thoroughly examined by three contract debugging experts. Almost 74% of the contracts in *dataset 1* support Solidity 0.5.x and the median value of the contract version is 0.5.2.
   - *Dataset 2*: This dataset comprises 4,744 anomalistic contracts covering 20 types of anomalies, which have not been manually verified. Although the contracts in dataset 2 have not been manually verified, according to our experimental results on dataset 1 (the contracts of which are randomly selected from dataset 2), more than 97% of the anomalies injected by *SCAnoGenerator* are accurate Almost 68% of the contracts in *dataset 2* support Solidity 0.5.x and the median value of the contract version is 0.5.3. Researchers may employ datasets 1 and 2 as the benchmark to assess the performance of analysis tools for anomaly detection.
   - *Dataset 3*: This dataset contains 66,208 *real-world contracts* without injected anomalies. Users can use the contracts in dataset 3 as the source for anomaly injection, that is, generating anomalistic contracts without worrying about *real-world contracts* collection. Almost 30% of the contracts in *dataset 3* support Solidity 0.5 x or higher and the median value of the contract version is 0.4.23.

3. Users can access these three datasets by visiting https://github.com/Our4514444/SCAnoGenerator-benchmark

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

2 BACKGROUND

This section introduces the basic concepts used in the paper. First, the concepts of Ethereum smart contract and Solidity are described in Section 2.1. Then, Section 2.2 introduces anomaly, bug and vulnerability. Additionally, smart contract anomaly detection criteria is introduced in Section 2.3. Finally, the 20 types of anomalies handled by *SCAnoGenerator* are detailed in Section 2.4.

2.1 Ethereum smart contract and Solidity

Users deploy smart contracts by sending contract bytecode to Ethereum through transactions. A transaction is a message that is sent from one account to another account (which might be the same or empty). In Ethereum, each contract or user is assigned a unique address as their identifiers. Ether is the default cryptocurrency of Ethereum. Both contracts and users can trade Ethers. Solidity [41] is the most widely used programming language for developing Ethereum smart contract. Users can employ Solidity to develop contracts. A compiler is then utilized to generate the contracts' bytecode. Solidity is a fast-evolving language. New versions of Solidity with breaking changes are released every few months. When developing a contract, developers need to specify the employed Solidity version, so as to use the proper compiler to compile the contract. Solidity assigns a *function selector* to each function. The *function selector* is the first four bytes of the *keccak-256* hash of the signature of the function, it is used to identify the function to be called. It is worth noting that in a few cases, different functions will be assigned the same *function selector* (i.e., hash collision). The overridden function has the same *function selector* value as the overriding function. Solidity supports (multiple) inheritance. Its official compiler (*solc*) can specify linear inheritance orders from base contracts to derived contracts. Solidity provides *require-statement* and *assert-statement* to handle errors. When the parameters of these two types of statements are *false*, *require-statement* or *assert-statement* will throw an exception and rollback the results of program execution.

2.2 Anomaly, bug and vulnerability

An anomaly refers to an unusual situation that does not conform to expected behavior or norms [42]. It can be caused by software errors, hardware failures, atypical user actions, or other unexpected factors. The scope of anomalies is relatively broad, which can include various situations that do not match the expected behavior. A bug refers to a known problem or defect in computer software or systems that prevents the program from working as expected. It can be caused by programming errors, logic errors, syntax error, or other implementation problems. Compared to an anomaly, a bug is usually caused by human error or negligence and have a more specific scope. A vulnerability refers to a flaw in computer software, systems, or networks that may be exploited by an attacker to gain unauthorized access, obtain sensitive information, or perform malicious operations [42]. Compared to anomalies and bugs, vulnerabilities involve system security and potential malicious exploitation [43], thus having a more specialized scope. Here we use anomaly as an umbrella term to encompass any unexpected behavior instead of bug or vulnerability in this paper.

2.3 Smart contract anomaly detection criteria

A number of smart contract anomaly/bug classification frameworks and corresponding detection criteria have been proposed [16], [44], [45], [46], [47], [48]. Among them, our previous work [45] proposes a comprehensive smart contract anomaly classification framework by extending the *IEEE Standard Classification for Software Anomalies* [48], which summarizes 49 types of bugs and their severity levels. Besides, this work [45] also proposes a set of anomaly detection criteria, i.e., a specific type of anomalies can be found in a contract as long as the contract matches certain characteristics. In this paper, we focus on how to inject the 20 most severe anomaly types, such as *re-entrancy* and *integer overflow and underflow*. Section 2.4 shows the name of each anomaly type. The specific description and anomaly detection criteria of each anomaly type can be found in [45].

2.4 Smart contract anomaly type

In this section, we describe the 20 types of anomalies targeted by our study.

Transaction order dependence. Miners (individuals or organizations that maintain the bitcoin network for the benefit of bitcoin, in terms of confirming transactions and package data) can decide which transactions are packaged into the blocks and the order in which transactions are packaged. This anomaly primarily impacts on the *approve* function in the *ERC20* token standard. It enables miners to influence the results of transaction execution. If the results of the previous transactions have an impact on the results of the subsequent transactions, miners can influence the results of transactions by controlling the order in which the transactions are packaged.

Results of contract execution affected by miners. A miner can control the attributes related to mining and blocks. If the functions of a contract depend on these attributes, the miner can interfere with the functions of the contract. This anomaly causes miners to gain a competitive advantage, which makes the contract's function inconsistent with the developer's expectations.

Unhandled false exception. A contract can use low-level call statements such as *send*, *call*, and *delegatecall* to interact with other addresses. When a call built upon these low-level call statements is abnormal, the call is not terminated and rollback. Instead only *false* is returned. This anomaly can cause a contract to be unable to learn and handle the exceptions generated during the call, which may affect the function of the contract.

Integer overflow and underflow. When a result exceeds the boundary value, the result will overflow or underflow. This anomaly can cause erroneous calculation results. If these calculation results are used to represent the amount of tokens (ethers), it will cause economic losses.

Use *tx.origin* for authentication. Solidity provides the keyword *tx.origin* to indicate the initiator of a transaction. It is not recommended to use *tx.origin* for authentication. When an attacker deceives a user's trust, the attacker can trick the user into sending a transaction to a malicious contract deployed by the attacker, and then the malicious contract forwards the transaction to the user's contract. At this point, the originator of the transaction is the user, so the attacker can be authenticated. This anomaly will allow the attacker to pass the identity verification, which may affect the function of the contract or cause economic losses.

Re-entrancy. When a *call-statement* is used to call other contracts, the callee can call back the caller and enter the caller again. This anomaly is one of the most dangerous smart contract anomalies, which will cause the contract balance (ethers) to be stolen by attackers.

Wasteful contracts. A contract that anyone can withdraw the ethers is called a *wasteful* contract. The reason for this anomaly is that the contract does not have authority control over the withdraw ethers. This anomaly can cause any user to take ethers from a contract, which would cause economic losses.

Short address attack. When Ethereum packs transaction data if the data contains the address type and the length of the address type is less than 20 bits, subsequent data will be used to make up the length of the address type. This anomaly may cause the attacker to withdraw tokens (ethers) equivalent to several times the number he requested.

Suicide contracts. Authority control must be performed before a self-destructing operation; otherwise, the contract can be easily killed by an attacker. This anomaly can cause the contract to be killed by anyone, and affect the function of the contract and cause economic losses.

Locked ether. If the contract can receive ethers, but cannot send ethers, the ethers in the contract will be permanently locked. This anomaly will cause all ethers in the contract to be permanently locked and will cause economic losses.

Forced to receive ether. An attacker can force ethers to be sent to an address through self-destructing contracts or mining. The attacker can forcibly send ethers to a contract. If the contract's function depends on the contract's balance being at a certain value, the attacker can use this error to affect or destroy the contract's function.

Pre-sent ether. Malicious users can send ethers to the address of a contract before the contract is deployed. If the function of the contract depends on the balance of the contract, then the pre-sent ether may affect the function of the contract. The consequences of this anomaly are the same as *Forced to receive ether anomaly*.

Uninitialized local/state variables. Uninitialized local/state variables will be given default values (eg., the default value of an address variable is 0x0, sending ethers to this address will cause ethers to be destroyed). This anomaly will increase the probability of developers making mistakes, which may eventually lead to the contract's function not being consistent with the developers' expectations.

Hash collisions with multiple variable length arguments. Because *abi.encodePacked()* packs all parameters in order, regardless of whether the parameters are part of an array, a user can move elements within or between arrays.

As long as all elements can cause different parameters to produce the same hash value, which may affect the function of a contract.

Specify *function* variable as any type. Function variables can be specified as any type through assembly code (up to and including version 0.5.16). This anomaly may cause a function variable to be assigned to any type, thus bypassing the necessary type checking. This will increase the probability of developers to make mistakes and may affect the functionality of a contract.

Dos by complex *fallback* function. If the execution of the *fallback* function consumes more than 2300 gas, sending ethers to the contract using *transfer-statement* or *send-statement* may fail. This anomaly can cause a contract to be unable to receive some externally transferred ethers (the ethers sent using the *transfer-statement* and *send-statement*), which may cause economic losses.

***Public* function that could be declared *external*.** Deploying a function with public visibility consumes more *gas* than deploying a function with *external* visibility. If a *public* function is not used in the contract, then declaring the function as *external* can reduce gas consumption.

Non-public variables are accessed by *public/external*. Solidity needs to specify the visibility of state variables, of which *internal* and *private* specify that state variables can only be accessed internally. However, using *public* or *external* functions to access *internal* and *private* state variables does not result in compilation errors. This anomaly causes users to access externally invisible state variables by calling *public/external* functions, which will affect the function of the contract.

Nonstandard naming. Solidity specifies a standard naming scheme. Following the standard naming scheme will make the source code easier to understand. Inconsistent or confusing variable names can result in developers referencing the wrong variables, increasing the likelihood of errors during execution. This anomaly will reduce the readability and maintainability of the source code. When developers review the code, they cannot quickly and accurately understand the type of each identifier.

Unlimited compiler versions. In different versions of Solidity, the same statement may have different semantics. When writing contracts, the Solidity version should be explicitly specified. In Solidity, the *pragma* keyword is used to enable certain compiler features or checks. It instructs the compiler to check whether its version matches the one required by the source code. This anomaly leads to a decrease in the maintainability of the contract. In future versions of Solidity, the same sentence may have different semantics. Therefore, not specifying the specific compiler versions will result in the inability to correctly understand the original meaning of the code when reviewing the code in the future.

3 *SCAnoGenerator*

3.1 Overview of SCAnoGenerator

SCAnoGenerator is an automatic anomaly injection tool for Ethereum smart contracts. It can inject 20 types of severe anomalies into the source code of *real-world contracts*. Specifically, *SCAnoGenerator*, extended from our previous work [45], employs heuristic algorithms to create anomalies in *real-world* contracts. It is worth mentioning that, similar to *SolidiFI*, *SCAnoGenerator* employs predefined templates to generate anomalies. The constructed anomalistic smart contracts can be used to evaluate both source code and bytecode based anomaly analysis tools.

The workflow of *SCAnoGenerator* is shown in Fig 1. *SCAnoGenerator* first collects *real-world contracts* (performed by *ContractSpider* introduced in Section 3.2). It then checks whether a contract is suitable for injecting a certain type of anomalies by analyzing the contract's control and data flows, and extracts the data required for injecting anomalies (achieved by *ContractJudgeAndExtractor* presented in Section 3.4). According to the data extracted by *ContractJudgeAndExtractor* required for anomaly injection, *SCAnoGenerator* injects anomalies into the contract (implemented by *AnomalyInjector* described in Section 3.5).

3.2 ContractSpider

The first step of *SCAnoGenerator* is to collect *real-world contracts* as the sources for anomaly injection so that users do not need to manually collect the contracts beforehand (of course, *SCAnoGenerator* also supports injecting anomalies into local contracts). We implement *ContractSpider* based on [49], which is a parallel high-performance web crawler. *ContractSpider* automatically collects contract source code by crawling open-source *real-world contracts* websites (e.g., http://etherscan.io/), and then saves these source code as Solidity files. Note that *ContractSpider* runs independently of *ContractJudgeAndExtractor* and *AnomalyInjector*. Users can run *ContractSpider* once to collect thousands of *real-world contracts*, and then users can generate the required anomalistic contracts by running *ContractJudgeAndExtractor* and *AnomalyInjector* according to their needs. In fact, Ethereum does not provide an interface to enable the direct access to all contracts. As a result, the repetitive crawling takes a lot of manpower and time. Given our limited time and resource, we collected (not selected) a total of 66,208 contracts (e.g., dataset 3., which is the biggest dataset in our experiment.

3.3 Construct a contract's control and data flows

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

Based on *solc* [50] and *Slither* [35], *SCAnoGenerator* constructs a contract's control and data flows. Specifically, *SCAnoGenerator* extracts all the function-call paths from a (derived) contract to track the definitions and use of data in each path. A function-call path refers to a sequence of nodes, each of which denotes a function. Adjacent nodes means that there is a one-way calling relationship between these functions (i.e., the former function calls the latter one).

- **Data flow**: Using *solc* to compile a contract can generate the abstract syntax tree (*AST*) of the contract. By analyzing *AST*, *SCAnoGenerator* obtains the following information: *definition-use* pairs [51] of data, and linear inheritance orders. Based on the above information, *SCAnoGenerator* tracks the definition and use of all the data.

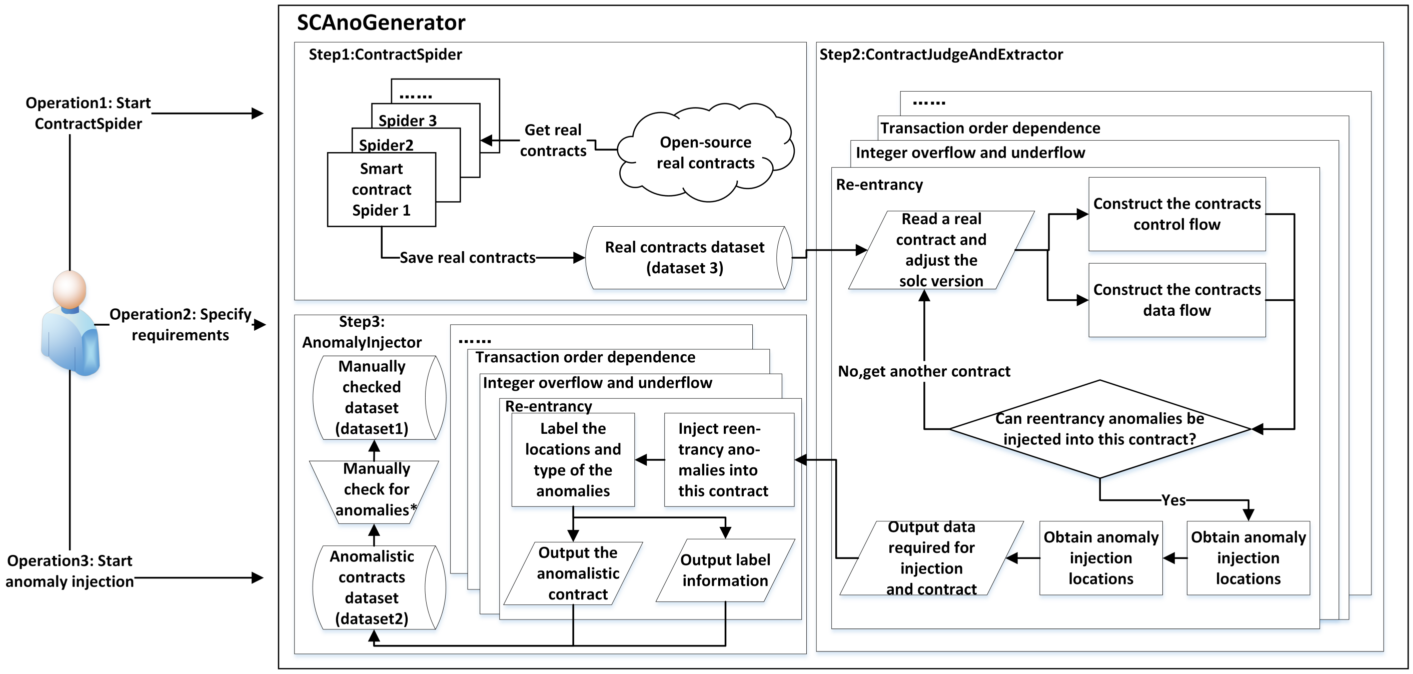
SCAnoGenerator
Step1:ContractSpider
Operation1: Start ContractSpider
Operation2: Specify requirements
Operation3: Start anomaly injection
Smart contract Spider 1
Spider2
Spider 3
......
Get real contracts
Open-source real contracts
Save real contracts
Real contracts dataset (dataset 3)
Step2:ContractJudgeAndExtractor
Transaction order dependence
Integer overflow and underflow
Re-entrancy
Read a real contract and adjust the solc version
Construct the contracts control flow
Construct the contracts data flow
No,get another contract
Can reentrancy anomalies be injected into this contract?
Yes
Obtain anomaly injection locations
Obtain anomaly injection locations
Output data required for injection and contract
Step3: AnomalyInjector
Manually checked dataset (dataset1)
Manually check for anomalies*
Anomalistic contracts dataset (dataset2)
Label the locations and type of the anomalies
Inject reentrancy anomalies into this contract
Output the anomalistic contract
Output label information

Fig. 1: *SCAnoGenerator* workflow, where a step name followed by * refers to a manual process. *SCAnoGenerator*(SG) contains three steps. As described in Section 3.2, Step 1 (ContractSpider) is to crawl a real smart contract from the open source Ethereum platform. Step 2 (ContractJudgeAndExtractor) is to formulate different extractors according to 20 different anomaly types to determine what types of anomalies can be injected into the contract. Based on the result of Step 2, Step 3 (AnomalyInjector) is designed to inject corresponding anomalies. The injection mechanisms of two typical types of anomalies are described in Section 3.4 and 3.5, and the injection of the remaining 18 types of anomalies are summarized in Table 1.

Algorithm 1 Contract CFG construction algorithm

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

- **Control flow**: *SCAnoGenerator* generates a (derived) contract's *CFG* by using the control flow graph (*CFG*) of each function and the function-call graph generated by *Slither* as well as the linear inheritance order generated by *solc*. Algorithm 1 shows this process. When a function (or function modifier) overrides the same function (or function modifier) of the base contract, *SCAnoGenerator* uses the following two steps to identify the overridden function on the path and replaces it with the overriding function: (1) If the *function selectors* of these two functions are different, *SCAnoGenerator* will consider that there is no function override. If not, *SCAnoGenerator* enters step 2. (2) *SCAnoGenerator* reads the contract's source code, obtains the function headers (i.e., in the function definition, all parts before the function body are called the function header) of these two functions, and analyzes whether the headers are same. Inheritance exists in Solidity, which means that a subclass may override the method inherited from its parent class. In this case, *SCAnoGenerator* considers that the derived class function overrides the base class function.

After constructing the function-call paths, *SCAnoGenerator* needs to ensure that each path is feasible, which can improve *SCAnoGenerator*'s anomaly injection accuracy. However, using the existing program path feasibility detection technology (e.g., *symbolic execution* [52]) to check whether each path is feasible will bring a huge performance burden and greatly reduce *SCAnoGenerator*'s anomaly injection speed. Considering that *SCAnoGenerator*'s design purpose is to help users build large-scale datasets of smart contracts with diverse anomalies, *SCAnoGenerator* needs to balance anomaly injection speed and anomaly injection accuracy. Therefore, we choose a compromise solution. Some studies [53], [54], [55] on the feasibility of program paths point out that interrelated conditional statements are the main reason for the infeasibility of program paths. Therefore, *SCAnoGenerator* replaces the conditional part of each branch conditional statement on every path with an ever-true expression to ensure the feasibility of the path. For the reentrancy anomalies in the labeled dataset, the number of ever-true expressions accounts for only 0.7% of conditional statements. We calculated the proportion of the number of lines of conditional statements that have been modified to ever-true expressions in the total number of lines of contracts that have been injected with re-entrancy anomalies, which is about 0.7%. It is worth noting that these ever-true expressions are randomly selected from a pre-prepared collection and replacing conditional statement with ever-true expression may affect the data flow or control flow of the original smart contract to a certain extent. Meanwhile, to ensure the difficulty in solving these ever-true expressions, and to prevent some dynamically executed tools from easily skipping the workload of condition judgment that should have been executed, these expressions are made to be of high complexity, involving several kinds of mathematical or logical operations.

3.4 ContractJudgeAndExtractor

One problem of injecting anomalies by inserting fully anomalistic code snippets into *real-world contracts* is that it is difficult to inject a large number of different complex anomalies. The reason is that triggering complex anomalies often requires multiple operations (e.g., variable declaration and initialization, variable state changes, and triggering anomalies) and thus it is not easy to write a large number of different code snippets containing complex anomalies. One way to solve this problem is to only inject a certain type of anomalies into the contract that is suitable for injecting that type of anomalies. For instance, the contract (*reentrancy*) shown in Listing 3 is a contract suitable for injecting *re-entrancy* anomalies. *SCAnoGenerator* only needs to construct a statement (e.g., the statement in line 16) and insert this statement before the statement in line 18 to inject a *re-entrancy* anomaly. Hence, *SCAnoGenerator* first checks if a contract is qualified for being injected with a certain type of anomalies.

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

Listing. 3:A contract that *SCAnoGenerator* can inject a *re-entrancy* anomaly.

ContractJudgeAndExtractor is designed to achieve this goal. *ContractJudgeAndExtractor* takes in a contract collected by *ContractSpider*, extracts the contract's control and data flows, and analyzes the control and data flows to check whether a certain type of anomalies 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 anomalies, *ContractJudgeAndExtractor* will pass the contract and the data required for anomaly injection to *Anomaly-Injector*; otherwise *ContractJudgeAndExtractor* will skip this contract. It is noteworthy that for different types of anomalies the internal workflows and the employed techniques of *ContractJudgeAndExtractor* are distinct. In this paper, we use two representative anomalies, namely *re-entrancy* and *integer overflow and underflow*, as examples to describe in detail how *ContractJudgeAndExtractor* works, and we briefly introduce *ContractJudgeAndExtractor(s)* of the remaining 18 types of anomalies in Table 1.

3.4.1 *ContractJudgeAndExtractor for re-entrancy (CER)*

According to [14], *re-entrancy* anomalies refer to reentrant function calls which divert money in an unexpected way. From *The DAO* [56] accident, a typical form of *re-entrancy* anomalies is that the *call-statement* that does not specify a response function is used to send ethers to the payee, and the *call-statement* is executed before the *deduct-statement* (i.e, the statement that deduct the number of tokens held by the payee address). This allows the payee to withdraw several times the same amount of ethers that he(she) deposited into the contract and even drain the contract's ethers. Since *SCAnoGenerator* can insert the *call-statements* that do not specify a response function to send ethers to payee , the key to injecting *re-entrancy* anomalies is to find the *deduct-statements*. To achieve it, *SCAnoGenerator* first locates the variables that record the relationship between the addresses and the number of tokens held by the addresses (we call such variables as *ledgers*).

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

- **Step 1**: *CER* searches for the *deposit paths* in the contract. The *deposit path* refers to a function-call path in the contract meeting the following conditions: 1) The entry function of this path is a *public* function declared as ***payable***. 2) There is at least one value increment operation on a *mapping(address=>uint256)* variable in this path. We call the *mapping(address=>uint256)* variable as a *potential ledger*. We call the set of all the *potential ledgers* in the contract as a *potentialLedgerSet*.
- **Step 2**: *CER* searches for *withdrawal paths* in the contract. The *withdrawal path* refers to a function-call path in the contract meeting the following conditions: 1) There is at least one value decrement operation on a *potential ledger* in this path. We call a *potential ledger* that meets this condition as a *target ledger*. 2) There is at least one operation to send ethers in this path, where the payee address needs to be the 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

TABLE 1: The *ContractJudgeAndExtractor(s)* and *AnomalyInjector(s)* of the remaining 18 types of anomalies

Anomaly type	**ContractJudgeAndExtractor**	**AnomalyInjector**
Transaction order dependence	The contract (**Con**) needs to be developed based on *ERC-20* token standard and contains the *approve* function	*SCAnoGenerator* **(SG)** invalidates security measures to allow the quota of the approved address to be set from one nonzero value to another nonzero value, and labels the assignment statement as an anomaly
Results of contract execution affected by miners	**SG** searches for the *if-statements* in **Con** that meet the following conditions: the condition part of the *if-statement* is one of the following types: *bytes32*, *address payable*, *uint256* or *address*	**SG** replaces an operand in the condition part of the *if-statement* with the following global variables: *block.coinbase* (address type), *block.coinbase* (address payable type), *block.gaslimit* (uint256 type), *block.number* (uint256 type), *block.timestamp* (uint256 type), *blockhash(block.number)* (bytes32 type), and labels the *if-statement* as an anomaly
Unhandled false exception	**Con** needs to contain at least one of the *low-level call statement* (i.e., *call-statement*, *send-statement*, and *delegatecall-statement*)	**SG** uses the following 2 ways to invalidate the security measures of the low-level call statement: receiving the return value but not checking the return value, or not receiving the return value. **SG** labels the *low-level call statement* as an anomaly
Use *tx.origin* for authentication	**SG** captures the *address type* variables assigned in the *constructor* function (we call these variables *ownerCandidate*) and then searches for *bool* expressions such as (*ownerCandidate ==(!=) address type variable*) in the contract. **Con** needs to contain at least 1 *bool* expressions that meets the above conditions	**SG** replaces *address type* variable (non-*ownerCandidate*) in the *bool* expression that meets the extraction criteria with *tx.origin*, and labels the *bool* expression as an anomaly
Wasteful contracts	**Con** needs to contain functions with the following conditions: the function's visibility is *public* or *external*, and the function needs to contain the statements that can transfer out ethers	**SG** invalidates all statements in the above functions (including the function modifiers) that may interrupt the execution of these functions, and inserts a statement to transfer out all ethers of **Con** at the end of the function (e.g., *msg.sender.transfer(address (this.balance);)*). Finally **SG** labels the statement that can transfer out ethers as an anomaly
Short address attack	**Con** needs to contain functions that meet the following conditions: 1) the visibility is *external* or *public*, 2) the function needs to contain statements that can transfer out ethers, and 3) the payee address and the number of ethers transferred are provided by the function caller	**SG** invalidates all check statements (i.e., statements that check *msg.data.length*) in the above functions (and the function modifiers), and labels these functions as anomalies
Suicide contracts	**Con** needs to contain functions that meet the following conditions: visibility is *external* or *public*, and functions need to contain *self-destruct-statements* (e.g., *suicide-statement* and *selfdestruct-statement*)	**SG** invalidates all statements in the above functions (and the function modifiers) that may interrupt execution or authenticate, and labels the *self-destruct-statements* as anomalies
Locked ether	**Con** needs to contain functions declared as *payable*	**SG** invalidates all the statements used to transfer out ethers in **Con** in the following 2 ways: setting the number of transferred out to 0, or changing the *transfer-out-statements* (i.e, *send-statement*, *transfer-statement*, *call-value-statement*) to comments, and label the contract as containing this type of anomalies
Forced to receive ether	**Con** needs to contain at least 1 of the 3 types of statements: *if-statement*, *require-statement*, or *assert-statement* that meet the following conditions: the type of the condition part of the statement is *uint256*, and at least one operand in the condition part is *uint256* literals	**SG** will replace the *non-uint256-literals* operand in the condition part of the above statements with *address(this).balance*, and label these statements as anomalies
Pre-sent ether. According to [45], *Forced to receive ether* anomaly and *Pre-sent ether* anomaly are the manifestations of the same type of anomalies in different periods. Therefore, the *ContractJudgeAndExtractor(s)* of these 2 types of anomalies are the same. The difference is that the anomaly types labeled by the *AnomalyInjector(s)* of the 2 types of anomalies are different		
Uninitialized local/state variables	**Con** needs to contain initialization statements for local variables or state variables (non-constant-varibales)	**SG** invalidates the assignment part in the initialization statement (e.g., *uint num; //= 1;*), and labels the initialization statement as an anomaly
Hash collisions with multiple variable length arguments	**Con** needs to contain functions that meet the following conditions: 1) visibility is *external* or *public*, 2) functions contain multiple array type parameters, and 3) function contains the declaration statement for the *bytes32* variable	**SG** re-assigns the *bytes32* variables in the above functions to *keccak256(abi.encodePacked(para))*, where the *para* are the array parameters passed by the external callers, and then labels the *bytes32* assignment statement as an anomaly
Specify *function* variable as any type	**Con** needs to contain *function* type variables	**SG** inserts *assembly-statements* so that external attackers can modify the type of *function* variables and then labels the inserted *assembly-statements* as anomalies
Dos by complex *fallback* function	**Con** needs to contain the *fallback* functions that meet the following condition: the *fallback* function contains statements that can transfer out ethers	**SG** inserts the statements at the end of the above *fallback* functions (e.g., *payee address.call.gas(2301).value(1)("");*), where the *payee* address is the payee address in *fallback* function. Then **SG** labels the *fallback* functions as anomalies
Public function that could be declared *external*	**Con** needs to contain functions with *external* visibility	**SG** changes the visibility of the above functions from *external* to *public* and labels these functions as anomalies
Non-public variables are accessed by *public*/*external* function	**Con** needs to contain *external/public* functions	**SG** searches for whether the above functions contain access operations to *private/internal* state variables with visibility. If so, **SG** will label these access operations as anomalies
Nonstandard naming	**Con** needs to contain at least 1 of the following 4 types of structures: *function*, *function modifier*, *event* and *constant* variable	**SG** changes the naming of these structures to non-standard form, and then modifies the naming of these structures in **Con**. Then **SG** labels the declaration statements of structures as anomalies
Unlimited compiler versions	**Con** needs to contain the *pragma-solidity-statement*	**SG** analyzes the *pragma-solidity-statements* to get the oldest Solidity version that can compile the contract, and then replaces the original *pragma-solidity-statements* with the following statement: *pragma solidity* $\wedge$ *minimum Solidity version*. And **SG** labels the new *pragma-solidity-statements* as anomalies

locations for all the value decrement operations in the *withdrawal path* as a *deductSet*.
- **Step 3**: All variables in the *ledgerSet* can be regarded as the *ledgers* in the contract and the *deductSet* that records the locations of all *deduct-statements*.

If a contract's *deductSet* is not empty, *SCAnoGenerator* only needs to insert the *call-statements* for sending ethers in front of all the locations in the *deductSet* to inject *re-entrancy* anomalies into the contract. For instance, in the contract shown in Listing 3, 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}. *SCAnoGenerator* only needs to insert a *call-statement* for sending ethers in line 16 to inject a *re-entrancy* anomaly.

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

3.4.2 ContractJudgeAndExtractor for integer overflow and underflow (CEI)

According to [45], the characteristics of the *integer overflow and underflow* (*IOA*) anomaly are: *Characteristic 1*: The maximum (or minimum) values that are generated by the operands participating in the integer arithmetic statement can exceed the storage range of the result. *Characteristic 2*: The contract does not check whether the result is overflow or underflow.

To construct the above characteristics, *CEI* needs to find integer arithmetic statements fulfilling the following conditions in a contract: *Condition 1*: The types of the operand variables and the result variable are same. This condition can ensure that the statement meets *characteristic 1*. *Condition 2*: In the integer arithmetic statement, at least one operand variable is a parameter passed by the external caller. This condition can ensure that the injected anomalies can be exploited by external attackers.

```
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 }
```

Listing. 4:How *SCAnoGenerator* invalidates security measures.

We call the integer arithmetic statements that meet *condition 1* and *condition 2* as *target statements*. As shown in Listing 4, when a *target statement* is found (line 14), *SCAnoGenerator* 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* anomaly. *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*.

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

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

When the *target* of a contract is not empty, *SCAnoGenerator* can inject *IOA* anomalies into the contract and pass the source code, *target*, and *opponent* of the contract to the *AnomalyInjector* of *IOA*.

3.5 AnomalyInjector

An *AnomalyInjector* receives a contract and the data required for injecting a specific type of anomalies from its corresponding *ContractJudgeAndExtractor* and performs anomaly injection and labeling. An anomaly injection tool always modifies the content of the contract. The less the anomaly injection tool modifies the contract content, the more the *injected contract* keeps its original structure. Hence, *AnomalyInjector* will only slightly modify the contract as much as possible.

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

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

used to check the contract state and roll back the transaction when the contract state is abnormal.

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

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

Listing. 5:How *SCAnoGenerator* constructs *call-statements* based on a payee address.

3.5.1 AnomalyInjector of re-entrancy (BIR)

BIR injects *re-entrancy* anomalies by inserting statements that cause anomalies. *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 Listing 5) and inserts the *call-statements* in front of the *deduct-statements* based on the *deductSet*. Finally, it labels the lines where the *call-statements* are inserted as having *re-entrancy* anomalies. For instance, in the contract shown in Listing 3, *SCAnoGenerator* will insert a *call-statement* (i.e., *_account.call.value(1)("")*) in front of the code in line 18, and label the inserted line as having a *re-entrancy* anomaly.

3.5.2 AnomalyInjector of integer overflow and underflow (BII)

BII injects *IOA* anomalies 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* anomalies. *opponent* contains the locations of *check statements*. *BII* invalidates all the *check statements* (by changing the statements to comments) in the *opponent* to make the statements in the *target* without security protection. Next, *BII* labels the statements in the *target* as having *IOA* anomalies. For instance, in the contract shown in Listing 4, *SCAnoGenerator* will change the code in line 5 to a comment (assuming the code in line 5 is not commented), and then label the code in line 14 as having an *IOA* anomaly.

4 Evaluation

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

- **RQ1**: Can *SCAnoGenerator* accurately inject anomalies ?
- **RQ2**: What is the usability of *SCAnoGenerator*?
- **RQ3**: Can users find more weaknesses in existing analysis tools by using *SCAnoGenerator*?
- **RQ4**: Can *SCAnoGenerator* efficiently inject anomalies?
- **RQ5**: Are the anomalies generated by *SCAnoGenerator* reliable?

4.1 Tools used in the evaluation

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

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

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

The state-of-the-art smart contract analysis tools are collected via two channels: 1) Analysis tools that have been covered by the latest empirical review papers, i.e., [27], [33]. 2) Analysis tools that are available on GitHub [57]. We use the keywords *smart contract security* and *smart contract analysis tools* to search in Github and select the twenty tools with the highest numbers of *stars* from the search results.

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

- *Criterion 1*. It supports command-line interface so that we can apply it to anomalistic contracts automatically.
- *Criterion 2*. Its input must be raw Solidity source code, rather than only considering EVM bytecode or other intermediate representation.
- *Criterion 3*. It is an anomaly detection tool, i.e., this tool can report types and locations of anomalies in contracts.
- *Criterion 4*. It can analyse contracts written in Solidity 0.5.x (i.e., 0.5.0-0.5.16). This is because *SolidiFI* can inject anomalies into contracts written in Solidity 0.5.0 and older versions, but *SCAnoGenerator* currently supports injecting anomalies into contracts written in Solidity 0.5.0 and subsequent versions. So, to ensure fairness, we will use the contracts that are written in Solidity 0.5.x for our evaluation.

According to the selection criteria, we finally select 6 analysis tools (*Maian* [38], *Manticore* [58], *Mythril* [36], *Securify* [39], *Slither* [35], *SmartCheck* [34]) to detect anomalies injected by *SCAnoGenerator* and *SolidiFI*. Table 2 lists the 6 selected tools.

4.2 Dataset for evaluation and environment

We use 3 types of datasets to perform this evaluation, e.g., labeled and unlabeled dataset, datasetθ. We first employed both *SCAnoGenerator* and *SolidiFI* to generate an *injected contract* dataset. The original *real-world contracts* were sourced from the Solidity 0.5.x contracts in *dataset 3* described in the introduction.

We then evaluated the anomaly injection performance of the two tools by inspecting each contract in the *injected contract* datasets. To ensure the fairness, we used *SCAnoGenerator* and *SolidiFI* to inject the same types of anomalies into the same contracts to generate the dataset. In other words, two contracts were generated based on each original contract respectively by *SCAnoGenerator* and *SolidiFI*. It is worth noting that, unlike *SolidiFI*, *SCAnoGenerator* cannot inject arbitrary types of anomalies into a contract because *SCAnoGenerator* only can inject a certain type of anomalies into the *real-world contracts* that meet the predefined extraction criteria. Therefore, for each of the 7 types of anomalies that can be injected by both *SCAnoGenerator* and *SolidiFI*, we first used *SCAnoGenerator* to generate 50 *injected contracts*, and then utilized *SolidiFI* to inject the same anomalies into the same group of contracts. For each of the 13 types of anomalies that can only be injected by *SCAnoGenerator*, we used *SCAnoGenerator* to generate 50 *injected contracts*. Finally, we obtain a dataset (i.e. **the labeled dataset**) comprising 896 *injected contracts* generated by *SCAnoGenerator* and 322 *injected contracts* generated by *SolidiFI*. The incomplete *injected contract* dataset results from the fact that *SCAnoGenerator* could not find 50 qualified contracts from dataset 3 for certain types of anomalies. Apart from the labeled dataset, we also generate an **unlabeled version** of the dataset (i.e., *injected contracts* without anomaly labels) based on the labeled dataset for anomaly injection accuracy and success evaluation of *SCAnoGenerator* and *SolidiFI*. We combined both analysis tools and human experts for this evaluation. This is because the existing analysis tools cannot ensure 100% anomaly detection accuracy (which also results from our motivation to design an anomaly injection tool that can stimulate the current anomaly detection research). These debugging experts did not participate in the design and development of *SCAnoGenerator*. After completing SG and uploading it to Github, we advertised the call for participation on our lab website. After receiving a number of applications and conducting the interview, we selected 3 debugging experts. All of them are skilled smart contract developers who have an average of 3-4 years of experience in smart contract development. A total of 3 experts participated in the evaluation, which took about three months. The debugging session of human experts contains two main parts. First, after we used *SCAnoGenerator* and *SolidiFI* to inject anomalies, we provided the injected but unlabeled contracts to these experts to check if those contracts are consistent with the basic facts, that is, whether the contracts contain anomalies. If the answer is yes, they need to indicate which lines of code have anomalies. Second, the annotated dataset was provided to the experts to evaluate whether the annotations are reasonable and validated according to the locations of anomaly injection. Each expert's evaluation is independent here. More specifically, among the three reviewers who checked the injected anomalies, when at least two reviewers reported that an anomaly is reasonable, we considered that the anomaly injection is effective. At the same time, we used Fleiss's Kappa (an extension of Cohen's Kappa) to evaluate the consistency among the experts' opinions. Considering the workload issue, we randomly selected 50 contracts' inspection notes from the dataset (labeled dataset and datasetθ) for statistics with a score of 0 or 1, where 0 represents that the anomaly in this contract is invalid and 1 represents that the anomaly in this contract is valid. After the score collection, we compiled the results into a two-dimensional table (120 rows * 2 columns), where 120 rows represent the number of anomalies that can be injected into 50 contracts, and 2 columns represent how many experts give a score of 0 or 1 to a single anomaly in each row. The Fleiss Kappa coefficient is 0.89, which indicates that almost perfect agreement among these experts' opinions. To assess the usability of *SCAnoGenerator*(e.g., how generic the used extraction criteria are for each anomaly type), we expand the number of contracts to be injected. datasetθ is a collection of 1,000 contracts coded in Solidity version 0.5.x randomly selected from *dataset* 3. To sum up, we performed experiments on labeled and unlabeled datasets to answer RQ1, RQ3, RQ4 and RQ5. We performed experiments on datasetθ to answer RQ2. These three datasets support Solidity 0.5.x. The median versions of the labeled or unlabeled dataset and the datasetθ are 0.5.2 and 0.5.12 respectively. Other details can be found in Table 3.

To prove that our datasets (i.e., datasets 1,2 and 3) represent "real-world" contracts [40], we use box-plots to describe the distribution of received transactions per contract among the datasets in Fig 2, where the y-axis is the logarithm of the number of received transactions per contract. It can be observed that the distribution is consistent among these datasets. Due to relatively small numbers of transactions, the width of the box charts of dataset 1 and 2 is slightly wider than that of dataset 3. 94.7% of contracts receive about 8 transactions, which is consistent with Oliva et al. [40]'s finding that 94.7% of contracts receive less than 10 transactions. There are also a certain number of abnormal values in the figure. This is because there are a few contracts that can receive up to 10,000 of transactions. Besides, we used Google BigQuery to calculate the creation date of a contract (we view the timestamp of a block where a contract was created as the creation date of the contract). Specifically, we use the following SQL statement SELECT DATE(block_timestamp) FROM 'bigquery-public-data.crypto_ethereum.contracts' WHERE address='xxx' for querying the time of contract creation, where xxx is replaced with the address of a contract in these datasets. Since the time span of contracts in these three datasets is relatively large, we calculate the creation date of contracts based on a 6-month interval. The time of contract creation in the dataset starts from July 2018 to December 2020. The distribution of the contracts in each interval is also shown in Table 3.

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.

TABLE 3: Contract benchmark. Num represents the the number of contracts in the following dataset. LOC represents the number of lines of code (including comments), F represents Functions, M represents Function Modifiers and Transactions represents received transactions (average per contract). Contract creation date represents timestamp of a block where a contract was created. For data (i.e., LOC, F+M and Transactions) containing two columns, the first column refers to an average value and the second column refers to a median value.

Dataset	Num	LOC		F+M		Transactions		Contract creation date				
								Jul-Dec 18	Jan-Jun 19	Jul-Dec 19	Jan-Jun 20	Jul-Dec 20
Dataset 1	964	545	318	54	37	3,107	8	2.40%	28.37%	2.40%	21.15%	45.67%
Dataset 2	4,744	565	351	52	38	4,202	8	3.90%	29.50%	5.40%	20.60%	40.60%
Dataset 3	66,208	376	205	35	24	5,878	9	4.10%	27.43%	6.78%	25.70%	35.99%
labeled	896	673	334	65	37	3,267	7	2.50%	25.70%	5.50%	22.00%	44.30%
datasetθ	1,000	566	327	52	38	1,620	7	4.00%	26.40%	8.40%	23.50%	37.70%

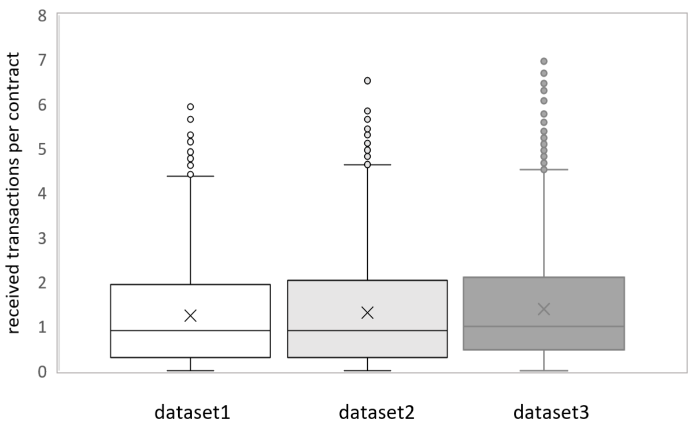

Fig. 2: The distribution of received transactions per contract of dataset 1, 2 and 3

4.3 RQ1: Anomaly injection accuracy

We calculate the accuracy of anomaly injection as follows:

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

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

- whether the *injected contract* can be compiled. Since the anomaly 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 anomalies in the contract are not deemed to be activated.
- whether the injected anomalies can be exploited by attackers and cause the expected consequences. Three smart contract debugging experts manually check each injected anomaly on the premise of knowing the locations and types of injected anomalies, and reach a consensus through discussion. If an anomaly cannot be exploited, it cannot be activated.

Result. In this experiment, we used the unlabeled version of the dataset. A contract that cannot be compiled indicates that the contract cannot be compiled to generate corresponding ast files through running the corresponding version of smart contract compiler *solc*. After making detailed statistics, we can see that the anomalistic datasets generated by *SCAnoGenerator* can be compiled by *solc* to generate corresponding ast files. In contrast, not all anomalistic datasets generated by *SolidiFI* can be compiled by solc. For example, among the 322 contracts, for *Results of contract execution affected by miners*, 3 contracts failed to compile. *SolidiFI* fails to compile in case a variable is not defined but the variable is used. Of the 498 re-entrancy anomalies injected by *SolidiFI*, only a small amount over 100 can be validated by experts (See one of analysis in Fig 1). Some extremely rare security measures (e.g., only one re-entrancy anomaly injection of one contract cannot be validated by human experts due to ReentrantLock) prevent the injected anomalies generated by SCAnoGenerator from being validated by our human experts. Of the 1623 transaction order dependency anomalies injected by *SolidiFI*, almost 405 cannot be validated by experts (See one of analysis in Fig 6). Table 4 shows the *accuracyRate*(s) of *SCAnoGenerator* and *SolidiFI* for anomaly injection, where *N/A* means that the anomaly injection tool is not designed to inject a type of anomalies. We can see that *SCAnoGenerator* can inject more types of anomalies with higher accuracy, compared with *SolidiFI*. Specifically, *SCAnoGenerator* shows **equal or higher** *accuracyRate(s)* than *SolidiFI* for all the 7 comparable types of anomalies.

```
bool claimed_TOD22 = false;
address payable owner_ToD22;
uint256 reward_TOD22;
function setReward_TOD22() public payable{
require(!claimed_TOD22);

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

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

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

Listing. 6:A *transaction order dependence* anomaly injected by *SolidiFI* that cannot be activated

Analysis. By interviewing the debugging experts, we obtain the main reasons for the injection failures of *SolidiFI* and *SCAnoGenerator*. The main reason for the injection failures of *SolidiFI* is that the inserted code snippets contain dead code. e.g., Listing 6 shows a sample code snippet inserted by *SolidiFI*. The original purpose of the code snippet is to inject a *transaction order dependence* anomaly into the contract by switching the execution order between two functions *setReward_TOD22* and *claimReward_TOD22*. If *claimReward_TOD22* is executed before *setReward_TOD22*, the statement in line 5 will throw an exception. However, since *SolidiFI* does not assign a suitable value to the variable *owner_TOD22*, it causes that *owner_TOD22* keeps the default initial value (*0x0*) and the *require-statement* in line 7 throws an exception. This eventually results in the statements in lines 8 and 9 turning into dead code and the executing function *setReward_TOD22* becoming useless.

The main reason for the injection failures of *SCAnoGenerator* is the excessive security measures used in some contracts. In some contracts, developers use multiple redundant security measures to prevent anomalies. Although *SCAnoGenerator* can invalidate most of the common security measures (we obtain the common security measures for each type of anomalies through extensive investigations), some extremely rare security measures prevent the injected anomalies from being activated. As the code in the Listing 7 shows, now we insert an anomaly statement on Line 7. Line 1 declares a *wallet* variable of mapping type. Line 2 declares *flag*, and its default value is true. Under normal circumstances, the money transfer operation can only be performed when the flag is true. Before the money transfer, the flag is changed to false. Only after the money transfer is completed, the flag is reset to true. In fact, transfer has already played the role of a protection contract, but the declaration of *flag* has played the role of repeated protection. We call *flag* a reentrancy lock. Now imagine a scenario: when the attack contract calls this code segment, the money transfer operation of Line 7 will transfer the ownership of the contract to the external contract, but the *flag* at this time is still false. Therefore, when calling the *tranfer* function again, the *flag* is equal to false and cannot enter the function, resulting in invalid anomaly insertion.

```
1  mapping (uint => address) wallet;
2  boolean flag = true;
3  function withdraw(){
4  require(flag == true);
5  flag = false;
6  require(wallet[address1] > 0);
7  address1.call.value(1 wei);//Injected anomaly
8  wallet[address1] -= 1 wei;
9  flag = true;
10 }
```

Listing. 7:A *reentrancy* anomaly injected by *SCAnoGenerator*

4.4 RQ2: Usability assessment

SCAnoGenerator relies on predefined extraction criteria for each anomaly type to decide if a contract is qualified for being injected with a certain type of anomalies. In this section, we carry out usability assessments to understand the capability of *SCAnoGenerator* to inject different types of anomalies into contracts, so as to understand the generalizability of the extraction criteria for each anomaly type. We evaluate the usability of our method from two aspects: 1) how many anomalies (on average) can be injected in each contract for each anomaly type; and 2) given a certain number of contracts to be injected, how many contracts meet the conditions for qualified injection of certain types of anomalies?

We assess Aspect 1) in the following way:

$$averageAmount = (BIN - IABN) \div CIN \tag{2}$$

where *CIN* represents the number of contracts injected by the anomaly injection tool for each anomaly type, and *averageAmount* represents the number of anomalies (on average) can be injected in each contract for each anomaly type.

We measure Aspect 2) through the following metric:

$$successRate = CIN \div BCIN \tag{3}$$

where *BCIN* represents the total number of contracts to be injected for each anomaly type, and *successRate* represents the proportion of contracts meeting the injection conditions for each anomaly type. The lower the *successRate*, the more contracts required for anomaly injection, and more contracts excluded from anomaly injection.

Result. In this experiment, we use datasetθ. Table 4 shows the *averageAmount*(s) and *successRate*(s) of *SCAnoGenerator* for anomaly injection. For *averageAmount*, we can see that at least one anomaly is injected in each contract for each anomaly type. For some anomaly types, such as *Nonstandard naming* and *Non-public variables are accessed by public/external*, using *SCAnoGenerator* can often inject the anomalies in multiple places of a contract. For *successRate*, we find that most contracts are excluded for anomaly injection for some anomaly types like *Transaction order dependence*, due to the effect of the predefined extraction criteria.

Analysis. We analyse the generalizability of the anomaly injection criteria set by *SCAnoGenerator* from the aforementioned two aspects.

When evaluating how many anomalies (on average) are injected in each contract for each anomaly type (i.e., *averageAmount*), we employ *SCAnoGenerator* to inject anomalies to create a dataset with labels. *SCAnoGenerator* is based on the analysis of the data flow and control flow of an injected contract combined with the structure of the contract to inject anomalies instead of arbitrary injection. It is found that the selected contracts are with diverse lengths. *SCAnoGenerator* needs to find the proper places from each contract to inject anomalies according to the predefined extraction criteria. It can also be computed based on the result of Table 4 that, for certain types of vulnerabilities, about 85% of contracts can be injected with more than 2 anomalies, and 47% of contracts can be injected with more than 7 anomalies.

When evaluating how many contracts are included for anomaly injection (i.e., *successRate*), we screen *dataset* 3 to find Solidity 0.5.x-based contracts and randomly select 1000 contracts to form datasetθ. We then run *SCAnoGenerator* to perform anomaly injection. An anomaly can only be injected once the control flow and data flow of a contract are analyzed, and it is determined that the contract meets its corresponding injection conditions.

The main reasons why *SCAnoGenerator* excludes many contracts for anomaly injection and has a relatively low success rate are: 1) *SCAnoGenerator* has relatively strict standard to judge whether a certain type of anomalies can be injected into a contract. It employs the original structures of a contract (e.g., variables that have been declared and visible in the current scope) to construct the anomalistic statements to be inserted into the contract, which means that it is able to inject a type of anomalies into a contract

TABLE 4: The *captureRate*, *speed*, *averageAmount* and *successRate* of analysis tools when detecting anomalies injected by *SCAnoGenerator(SG)* and *SolidiFI(SF)*

Anomaly type	Injection tool	**accuracyRate**	**speed**	**averageAmount**	**successRate**
Transaction order dependence	*SG*	96.0%	34.1	1.0	0.7%
	SF	75.1%	0.5	29.2	26.7%
Results of contract execution affected by miners	*SG*	100.0%	1.2	6.2	28.9%
	SF	97.9%	1.7	30.6	33.5%
Unhandled false exception	*SG*	100.0%	3.4	1.7	6.6%
	SF	97.3%	0.6	30.6	33.1%
Integer overflow and underflow	*SG*	98.1%	4.0	4.6	10.1%
	SF	97.7%	1.2	28.4	37.2%
Use *tx.origin* for authentication	*SG*	100.0%	4.1	2.3	6.9%
	SF	81.9%	0.5	28.1	28.9%
Re-entrancy	*SG*	96.7%	352.1	1.0	0.2%
	SF	20.1%	0.4	29.3	7.7%
Wasteful contracts	*SG*	98.0%	5.6	1.5	6.2%
	SF	91.7%	0.8	24.7	27.8%
Short address attack	*SG*	100.0%	17.1	1.0	3.0%
Suicide contracts	*SG*	96.9%	209.8	1.0	3.0%
Locked ether	*SG*	100.0%	3.9	2.1	11.0%
Forced to receive ether	*SG*	100.0%	4.8	2.5	6.4%
Pre-sent ether	*SG*	100.0%	5.1	1.9	5.9%
Uninitialized local/state variables	*SG*	99.9%	0.7	15.3	25.1%
Hash collisions with multiple variable length arguments	*SG*	97.5%	84.0	3.9	0.5%
Specify *function* variable as any type	*SG*	100.0%	1949.8	1.3	0.3%
Dos by complex *fallback* function	*SG*	98.0%	61.2	1.2	1.0%
Public function that could be declared *external*	*SG*	100.0%	1.3	7.1	19.0%
Non-public variables are accessed by *public/external*	*SG*	98.7%	0.9	28.6	44.8%
Nonstandard naming	*SG*	99.8%	1.7	52.4	54.9%
Unlimited compiler versions	*SG*	100.0%	7.2	1.9	51.7%

only if the contract is recognized as eligible for injecting this type of anomalies. In other words, *SCAnoGenerator* cannot inject arbitrary types of anomalies into a contract and it cannot arbitrarily insert anomalies into contracts. For some anomaly types, such as those with low success rate of anomaly injection, the expected conductions are inherently uncommon in most contracts. For example, for the anomaly *specify function variable as any type*, *SCAnoGenerator* can inject this type of anomalies into a contract only if the contract contains function type variables. However, developers rarely use *function* type variables in contracts. Therefore, the *successRate* and *averageAmount* are relatively low. 2) The success rate of *SCAnoGenerator* is restricted by the cardinality of contracts qualified for being injected with each type of anomalies, so it is unwise to use fewer contracts to generate anomaly datasets, given the limited number of qualified contracts for each anomaly type. 3) We conducted an analysis on the datasetθ containing 1,000 contracts. After excluding contracts with zero transactions, the success rate of *SCAnoGenerator* is relatively high in the contracts with a high number of transactions, indicating the applicability of SG on popular contracts. For example, we found that almost 14% of the total contracts receive 1000+ transactions, where 66.3% of the contacts can be injected with anomalies. In contrast, 26.7% of the total contracts receive 100+ transactions, where 56.6% of these contracts can be injected with anomalies. In addition, the contracts that do not receive transactions can also be injected with anomalies, but the success rate is 10% lower than that of the contracts that receive transactions. Based on these findings, we reformulate the success rate calculation of SG, which is

$$successRate^{*} = (BIN - IBN) \div BIN \tag{4}$$

where *IBN* represents the number of anomalies that cannot be calculated for *successRate*. We calculated the value of *IBN* by checking the following three aspects, *IBN* has added Cat3 on top of *IABN*, the contracts in Cat3 are those that cannot receive transactions or have zero transactions.

- The injected contract cannot be compiled, which means all the injected anomalies in the contract are not deemed to be activated.
- The injected anomalies cannot be exploited by human experts.
- The contracts are not suitable for anomaly injection, which all the injected anomalies in the contract are not deemed to be activated.

We can see from the Table 5 that the average success rate is 64% for SG and the average success rate is 67% for SF. In addition, not all the anomalies injected by SG have a lower success rate than SF, such as *results of contract execution affected by miners* and *wasteful contracts*.

TABLE 5: The *successRate** of analysis tools when detecting anomalies injected by SG and SF

Anomaly type	Injection tool	**successRate***
Transaction order dependence	*SG*	50.0%
	SF	60.0%
Results of contract execution affected by miners	*SG*	77.9%
	SF	77.9%
Unhandled false exception	*SG*	71.2%
	SF	79.6%
Integer overflowand underflow	*SG*	76.2%
	SF	79.1%
Use *tx.origin* forauthentication	*SG*	76.9%
	SF	78.4%
Re-entrancy	*SG*	13.7%
	SF	15.7%
Wasteful contracts	*SG*	82.3%
	SF	78.7%
Short address attack	SG	83.3%
Suicide contracts	SG	74.4%
Locked ether	SG	80.0%
Forced to receive ether	SG	79.7%
Pre-sent ether	SG	81.4%
Uninitialized local/ state variables	SG	76.1%
Hash collisions with multiple variable length arguments	SG	72.5%
Specify *function* variable as any type	SG	70.1%
Dos by complex *fallback* function	SG	71.4%
Public function that could be declared *external*	SG	79.5%
Non-public variables are accessed by *public/external*	SG	77.0%
Nonstandard naming	SG	76.3%
Unlimited compiler versions	SG	76.8%

4.5 RQ3: Analysis tool based evaluation

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

$$captureRate = BDN \div BAN \tag{5}$$

where *BDN* represents the number of anomalies detected by an analysis tool, and *BAN* represents the number of anomalies that can be activated and injected by the anomaly injection tool. A lower *captureRate* indicates the weaker ability of an analysis tool to detect injected anomalies. Generally speaking, the output of the analysis tool is the type and location of the anomaly (in the form of line number or line number interval). *SCAnoGenerator* can provide the exact locations (i.e., line number) of the injected anomalies, and *SolidiFI* can only provide *loc* and *length* of its injected code snippet (the line number interval can be calculated by *loc* and *length*). Therefore, to count the injected anomalies captured by detection tools, we have developed capture criteria (as shown in Table 6). When the type and location of the injected anomaly reported by the detection tool meet the capture criteria, we will count the injected anomaly as a detected anomaly. We manually check the tools' documents to map the anomaly types that these tools can detect to the anomaly types that *SCAnoGenerator* 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 anomaly type. It is worth noting that the original anomalies in the *injected contracts*, i.e., those not generated by the injection tools, do not affect calculating the *captureRate*.

TABLE 6: Injected anomalies that meet the following criteria will be counted as detected anomalies

	Analysis tool	
	line number	line number interval
line number (SG)	line matching	contains this line number
line number interval (SF)	falls within this interval	subset

Result. This experiment is on the basis of the labelled dataset. Table 7 shows the *captureRate*(s) of *SCAnoGenerator* and *SolidiFI*. In Table 7, we omit the *captureRate(s)* of an anomaly injection tool in two cases: 1) This anomaly injection tool is not designed to inject a type of anomalies. 2) A type of anomalies injected by this anomaly injection tool cannot be activated. The experimental results of *realRate(s)* show which anomaly injection tool will have the above two cases when injecting which type of anomalies. * represents that an analysis tool is not designed to detect a type of anomalies. It can be seen from Table 7 that, compared to the anomalies injected by *SCAnoGenerator*, the anomalies injected by *SolidiFI* are easier to be detected by the detection tools. We randomly selected 20% of *injected contracts*, used 6 detection tools to detect these contracts again, and manually tracked their detection performance. The manual tracking process means that 20% of the contracts participating in this experiment are randomly selected to manually check the results of the detection tool, that is, whether the results of the tool conform to our basic fact of injecting anomalies (i.e. whether the locations of the anomalies detected by the tool are consistent with their actual locations in source code). The whole process is also done by the experts. We found that *Slither* achieved the best performance, which reports 97% of anomalies injected by *SolidiFI* and 68% of anomalies injected by *SCAnoGenerator*. However, it also produced a lot of false positives (56.2% of anomalies reported by *Slither* are false positives). *Mythril*, *Manticore*, and *Securify* all exceeded the timeout threshold when analyzing 65% of *injected contracts*. However, when their runtime is within the timeout threshold, they can detect 84% of the anomalies injected by *SolidiFI* and 47% of anomalies injected by *SCAnoGenerator*. These analysis tools can only detect at most 37.3% anomalies injected by *SCAnoGenerator*, which means that analysis tools cannot detect at least 62.7% anomalies injected by *SCAno-Generator*. In contrast, these detection tools only missed at

TABLE 7: The *captureRate* of analysis tools when detecting anomalies injected by *SCAnoGenerator(SG)* and *SolidiFI(SF)*

		captureRate					
Anomaly type	Injection tool	*SmartCheck*	*Slither*	*Mythril*	*Manticore*	*Maian*	*Securify*
Transaction order dependence	*SG*	0.0%	*	*	*	*	0.0%
	SF	0.0%	*	*	*	*	4.8%
Results of contract execution affected by miners	*SG*	7.4%	35.2%	1.8%	0.1%	*	*
	SF	55.8%	100.0%	5.4%	8.9%	*	*
Unhandled false exception	*SG*	26.2%	94.4%	0.2%	*	*	7.2%
	SF	25.8%	100.0%	6.7%	*	*	19.4%
Integer overflow and underflow	*SG*	*	*	1.0%	0.8%	*	*
	SF	*	*	15.6%	10.2%	*	*
Use *tx.origin* for authentication	*SG*	69.8%	72.6%	*	*	*	*
	SF	70.6%	95.6%	*	*	*	*
Re-entrancy	*SG*	*	100.0%	14.5%	0.0%	*	30.4%
	SF	*	96.4%	0.0%	0.0%	*	31.2%
Wasteful contracts	*SG*	*	63.1%	0.7%	*	0.0%	8.2%
	SF	*	98.4%	27.7%	*	0.0%	20.4%
Short address attack	*SG*	*	*	*	*	*	*
Suicide contracts	*SG*	*	76.3%	13.4%	0.0%	0.0%	*
Locked ether	*SG*	60.0%	78.0%	*	*	0.0%	0.0%
Forced to receive ether	*SG*	85.7%	99.4%	*	0.2%	*	0.2%
Pre-sent ether	*SG*	80.1%	99.4%	*	0.0%	*	0.0%
Uninitialized local/state variables	*SG*	*	71.9%	*	0.0%	*	*
Hash collisions with multiple variable length arguments	*SG*	*	*	*	*	*	*
Specify *function* variable as any type	*SG*	*	*	0.0%	*	*	*
Dos by complex *fallback* function	*SG*	*	*	*	*	*	*
Public function that could be declared *external*	*SG*	0.0%	77.6%	*	*	*	*
Non-public variables are accessed by *public/external*	*SG*	*	*	*	*	*	*
Nonstandard naming	*SG*	*	82.2%	*	*	*	*
Unlimited compiler versions	*SG*	100.0%	58.2%	*	*	*	0.2%

most 49.4% of anomalies injected by *SolidiFI*. 62.7% or 49.4% cannot be directly obtained from Table 7. We calculated the sum of *BAN* and *BDN* when its capture rate is the highest. For SG, the sum of *BAN* is 8,474, and the sum of *BDN* at its maximum capture rate is 3,161. For SF, the sum of *BAN* is 4,176, and the sum of *BDN* is 2,113 when reaching its maximum capture rate. The experimental results of *captureRate(s)* indicate that the use of *SCAnoGenerator* to evaluate the detection tools can reveal more weaknesses of these tools.

Analysis. We describe the performance of analysis tools in detecting the injected anomalies and propose possible directions for improvement.

The analysis tools based on pattern matching or feature capture, represented by *SmartCheck* and *Slither*, show **better** detection ability when detecting the anomalies injected by *SCAnoGenerator*. This is because the tools based on pattern matching or feature capture tend to achieve higher detection speed (so they hardly reach the time out threshold) and usually generate a lot of alerts. However, such tools also tend to generate a large number of false positives, which reduces the value of the generated detection results.

The analysis tools based on symbolic-execution, represented by *Mythril* and *Maticore*, show **weaker** anomaly detection ability. This is because these tools exceed the timeout threshold when analyzing many contracts. Although we set a long timeout threshold (15 mins), symbolic-execution tools need to take a much longer time to cover all the paths of the contracts due to the high complexity of the *injected contracts* generated by *SCAnoGenerator*. Besides, we found that even in the case of no timeout, it is difficult for these tools to find the complex anomalies (e.g., the *re-entrancy* anomalies caused by the interaction among multiple functions from multiple contracts). Our evaluation results are consistent with the evaluation results of existing detection tools [27], [59], [60], i.e., the performance of the current analysis tools is far from meeting the security requirements of smart contracts.

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

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

4.6 RQ4: Anomaly injection efficiency

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

$$speed = IT \div BIN \tag{6}$$

where *IT* represents the time it takes for the anomaly injection tool to inject anomalies. Note that we only count the aggregated running time of *ContractJudgeAndExtractor* and *AnomalyInjector* as *IT* rather than that of *ContractSpider* for *SCAnoGenerator*.

Result. Table 4 shows the anomaly injection *speed*(s) of *SCAnoGenerator* and *SolidiFI* (in seconds per anomaly). It can be seen that the anomaly injection speed of *SCAnoGenerator* generally **lags behind** that of *SolidiFI*.

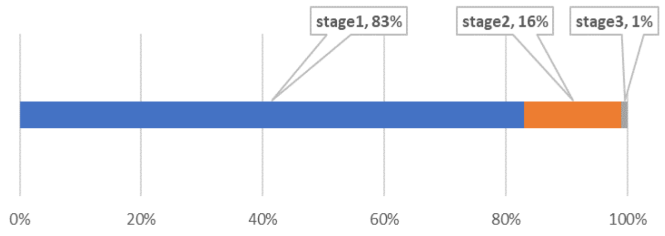

Fig. 3: The proportion of the time for each stage of the anomaly injection process

Analysis. We analyze the rationale for the lower anomaly injection speed of *SCAnoGenerator*. There are two reasons: 1) *SCAnoGenerator* spends substantial time running *solc* and *Slither*. It needs to go through three stages to inject a certain type of anomalies into a contract as aforementioned. Running *solc* and *Slither* to generate auxiliary information is the *first stage*. Constructing the contract's control and data flows and ascertaining whether the contract has a basis for injecting a certain type of anomalies is the *second stage*. Injecting a certain type of anomalies into the contract is the *third stage*. We measure the proportion of time taken for each stage. The results are shown in Fig 3. It can be seen that the running time of *solc* and *Slither* accounts for **most** of the running time of *SCAnoGenerator* (83%). 2) Contracts suitable for injecting different types of anomalies have different scarcity levels in Ethereum. For instance, *SCAnoGenerator* spends substantial time to inject a *specify function variable as any type* anomaly. This is because *SCAnoGenerator* can inject this type of anomalies into a contract, only if the contract contains *function* type variables. However, developers rarely use *function* type variables in contracts. According to the statistics, *SCAnoGenerator* needs to extract average 21,603 contracts to find a contract suitable for injecting this type of anomalies. In contrast, *SCAnoGenerator* only takes 209.8 seconds to inject a *suicide contracts* anomaly. This is because *self-destruct-statements* are more common than *function* type variables. *SCAnoGenerator* only needs to inject *suicide contracts* anomalies by invalidating the security measures of *self-destruct-statements*.

4.7 RQ5: Reliability of anomalies

This subsection is to verify the reliability of the anomalies generated by *SCAnoGenerator* in a real environment. We deploy the generated anomalistic contracts on *Remix* to replay the transactions and assess if the injected anomalies can be triggered. Transaction replaying refers to the input with specific conditions to see whether the contract will produce qualified output. Take the reentrancy in Listing 3 as an example. We construct the *Attack* contract in the Listing 8. First, we deploy the *reentrancy* contract to *Remix* and initialize the storage of 2 ether. Then we deploy the *Attack* contract and call the *deposit* function in line 8 to store 1 ether in the *reentrancy* contract. Now we want to withdraw the money we just deposited by directly calling the *withdraw* function on line 11 to withdraw the money. When the *reentrancy* contract transfers money to the *Attack*, the control of the contract will be transferred to *Attack*, and the *fallback* function on line 17 in *Attack* will be triggered. However, since *sendMoney* is called in the *fallback* function, all the balances in the original contract are forced to be transferred to *Attack*. By specifying practical input conditions, we verify whether all the places of a contract where anomalies are injected may trigger corresponding vulnerabilities. Considering the high workload, we randomly sample 100 contracts from *dataset* 1 by following the distribution of each type of anomalies in the dataset.

```
1  contract Attack{
2  reentrancy public reen;
3  mapping (address => uint256) balances;
4  constructor(reentrancy _reen) public payable {
5  reen = _reen;
6  balances[msg.sender] += msg.value;
7  }
8  function deposit() public payable{
9  reen.getMoney();
10 }
11 function withdraw() public{
12 reen.sendMoney(msg.sender);
13 }
14 function balanceOf(address depositor) public view returns(
       uint){
15 return balances[depositor];
16 }
17 fallback() external payable{
18 reen.sendMoney(msg.sender);
19 }
20 }
```

Listing. 8:An example of reentrancy anomaly attack contract.

It is worth noting that the whole process is to upload a contract onto *Remix* to carry on the execution of the contract rather than actually executing the contract in the chain environment. In Ethereum, transactions are the only way to trigger status changes. As it is a simulation experiment, we did not deal with the status of a contract or replay exact transactions. Moreover, we did not send real transactions to a contract. As for dealing with possible cross-contract calls, taking re-entrancy as an example (Listing 8), we put the calling contract (attacker) and the called contract on *Remix* at the same time to simulate and test contract anomalies.

Result. Table 8 shows the experimental result. The second column represents the number of contracts sampled for each anomaly type, The third column represents the number of contracts that can reproduce the corresponding vulnerability type, where * refers to no anomalistic contracts being deployed (due to the high computational power requirements). We found that **98%** of the generated contracts can trigger the corresponding anomalies in *Remix*.

Analysis. It can be seen that all the injected anomalies are successfully triggered except for the *Transaction order dependence* anomalies that are not deployed in *Remix* due to its high computational cost. Miners can decide which transactions are packaged into the blocks and the order in which transactions are packaged. Therefore, when performing experiments on this anomaly type, it requires an attacker to behave as a miner to monitor whether there are contracts on the network that depend on the transaction order. However, although it is revealed in the experiment, it is noteworthy that *SCAnoGenerator* may sometimes invalidate the injected anomalies. For example, for uninitialized local/state variables, *SCAnoGenerator* invalidates the assignment part in the initialization statement, which may cause the assignment in the *safeMath* library to be commented out. For Wasteful contracts, if a statement is inserted after the *selfdestruct* statement to transfer out all ethers (e.g., *msg.sender.transfer(address (this.balance)*, then the injected anomaly is also meaningless.

TABLE 8: The information of the sampling *dataset* from *dataset 1*

Anomaly type	samplingNum	reproNum
Transaction order dependence	5	*
Results of contract execution affected by miners	5	5
Unhandled false exception	5	5
Integer overflow and underflow	5	5
Use *tx.origin* for authentication	5	5
Re-entrancy	2	2
Wasteful contracts	5	4
Short address attack	5	5
Suicide contracts	4	4
Locked ether	5	5
Forced to receive ether	5	5
Pre-sent ether	5	5
Uninitialized local/ state variables	5	4
Hash collisions with multiple variable length arguments	4	4
Specify *function* variable as any type	1	1
Dos by complex *fallback* function	5	5
Public function that could be declared *external*	5	5
Non-public variables are accessed by *public*/*external*	5	5
Nonstandard naming	14	14
Unlimited compiler versions	5	5

5 Discussion

The motivation behind *SCAnoGenerator* is to evaluate the performance of existing anomaly detection tools through anomaly injection. In addition, its practical applications include the following aspects. Firstly, *SCAnoGenerator* can generate source code anomaly datasets, which can be used to conduct research on other smart contract security work. For example, it can guide code obfuscation work by simulating various attacks and analysis techniques, helping developers understand the effectiveness of obfuscation strategies in combating different types of attacks, and continuously improving obfuscation methods. To clarify, while *SCAnoGenerator* is primarily a fault injection tool, it can be effectively utilized to simulate certain attack scenarios by injecting specific vulnerabilities or malicious inputs into smart contracts. By doing so, it allows developers to observe how these injected faults can manifest into potential exploitative behaviors, thereby simulating the conditions under which an attacker might attempt to compromise a contract. Secondly, it can enhance smart contract testing. Testers use these tools to simulate the behavior of attackers and evaluate the robustness and quality of smart contract test cases. Thirdly, researchers can use *SCAnoGenerator* to explore and analyze new types of anomalies and their impacts, thereby advancing security research and potentially uncovering new vulnerabilities. Compared with *SolidiFI*, *SCAnoGenerator* has obvious advantages in accuracy and injection type. Firstly, *SCAnoGenerator* can inject more types of anomalies. Secondly, *SCAnoGenerator* can inject more accurate, authentic and logical anomalies. From the *SolidiFI* injection results, we can draw a conclusion that the anomaly injection of *SolidiFI* is not authentic. This is because only pre-defined code fragments are used by *SolidiFI* to insert fragments into a contract to complete the injection. There is no consideration about the original structure of the contract, which leads to the consequence that the anomaly fragments injected by *SolidiFI* are mostly non-logical and more likely to be detected. In contrast, *SCAnoGenerator* can more accurately inject suitable anomalies that are more difficult to be detected, which provides a more realistic and challenging environment to evaluate detection tools. However, *SCAnoGenerator* has limitations and threats. In this section, we discuss the Limitations and Threats to validity.

5.1 Limitations of *SCAnoGenerator*

The limitations of *SCAnoGenerator* originate from the following three aspects:

- *Restricted extensibility and limited types of injected anomalies*. So far we can employ *SCAnoGenerator* to inject 20 types of anomalies into smart contracts, which completely covers the 7 anomaly types that *SolidiFI* can inject. It is acknowledged that there are more types of anomalies in Ethereum. However, our research motivation is not to design an injection tool that can inject all available types of anomalies. Instead, we aim to develop a tool that can inject more types of anomalies that are more logical and difficult to detect. *SCAnoGenerator* anomaly injection starts with selecting contracts that meet the injecting conditions for specific anomalies. These conditions are based on our study on smart contract anomalies. Due to the limitation of our capacity, *SCAnoGenerator* may not inject anomalies covering all known real-world scenarios. To address this issue, we open the source code of *SCAnoGenerator* and generated datasets to promote collaboration and technical advance in this area. To support or extend a specific type of anomalies, we can analyze it based on the control flow graph constructed in Section 3.3, and formulate *ContractJudgeAndExtractor* and *AnomalyInjector* based on the control flow graph. For example, *byte[]* is a new anomaly type [45], which can act as a byte array, but it wastes 31 bytes of space for each element due to padding rules. Its solution is to use the bytes type instead of bytes[] for less gas consumption. When designing the *ContractJudgeAndExtractor* of *byte[]*, SCAnoGenerator searches for variables declared as byte in contract, and *AnomalyInjector* will replace *byte* type variable as *byte[]* and label this statement as an anomaly. During this process, developers need to write the code of *ContractJudgeAndExtractor* and *AnomalyInjector* to execute the injection of byte[]. To alleviate this issue, we plan to develop configuration files for vulnerability customization in the next phase. In addition, building the CFG can be modularized into a plugin. This plugin would handle the analysis of data flow and control flow within the contract, making it easier to integrate and extend with new types of anomalies. The 20 types of anomalies that we target in this project are the top-ranked anomalies in terms of severity and popularity. We believe that it can meet the general demand of most developers.

- *Restricted capacity or versions in generating injected contracts*. *SCAnoGenerator* is able to inject a type of anomalies into a contract only if the contract is recognized as eligible for injecting this type of anomalies. In other words, *SCAnoGenerator* cannot inject arbitrary types of anomalies into a contract, which means that, given many contracts to be injected, the *successRate* and *averageAmount* of *SolidiFI* injection are higher than those of *SCAnoGenerator*. This is a trade-off between authenticity and generation capacity for *injected contracts*. However, as long as there are enough *real-world contracts*, *SCAnoGenerator* can generate the required number of *injected contracts* with relatively high authenticity. In contrast, currently there is no dedicated external measures that can improve the *injected contract* authenticity. Due to timeliness issues, *SolidiFI* can support up to Solidity 0.5.12 and *SCAnoGenerator* can support Solidity 0.5.x, 0.6.x, 0.7.x. However, *SCAnoGenerator* is not fully compatible with Solidity 0.8.x and higher because its control or dataflow analysis module depending on *solc*. We have identified that most anomaly types listed in Table 1 are not affected by this version limitation. However, there are specific anomalies that are impacted. Firstly, *Integer overflow and underflow*. Starting from Solidity 0.8.x, all arithmetic operations (such as addition, subtraction, multiplication, and division) default to checking for overflow and underflow. If an overflow or underflow occurs, the operation will automatically revert. Because *SCAnoGenerator* invalidates security check require or assert to cause overflow and underflow, this injection may fail. In Solidity 0.8.x, integer overflow and underflow protection are enabled by default, which presents challenges for adapting *SCAnoGenerator* to this version. Secondly, *Nonstandard naming*. *ContractJudgeAndExtractor* of *SCAnoGenerator* needs to contain at least one of the following four types of structures: *function*, *function modifier*, *event* and *constant* variable. Starting from Solidity 0.8.x, constant is replaced by immutable and constant to distinguish between immutable and constant variables. This change may result certain immutable variables not being captured, causing injection failure. To adapt to this version, the tool needs to include the *immutable* keywords on top of the above four types.
- *Inadequate path reachability proof*. Considering whether each path can be reached after anomaly injection, we design a compromise method, which is to use ever-true expression to replace conditional statements in Section 3.3. We quantified this threat by the proportion of statements modified by ever-true statements and provided the replacement rates of the 20 types of anomalies injected by SCAnoGenerator in Table 9. In a contract, *ST* represents the total number of statements modified by the ever-true statements, and *TC* TC represents the total number of conditional statement branches. The calculation formula for *replaceRate* is shown in the formula 7. From this Table 9, we can see that very few conditional judgment statements are replaced. Therefore, the threat caused by ever-true statement replacement is relatively minor.

$$repalceRate = ST \div TC \tag{7}$$

5.2 Threats to Validity

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

The potential threats to **internal validity** is that *SCAnoGenerator* can only inject 20 types of anomalies. Apart from the 7 types of bugs supported by *SolidiFI*, we include 13 additional types of anomalies by thoroughly evaluating their severity and universality. Therefore, we believe that the selected 20 types of anomalies are representative and critical. There are more types of anomalies in Ethereum, which are not supported by the current version of *SCAnoGenerator*. To eliminate this threat, we plan to expand *SCAnoGenerator* to cover more anomaly types. We also aim to design machine learning based solutions to support data driven anomaly learning and injection in the future.

The potential threats of **external validity** results from three aspects. First, there are *toy/ education /test contracts*

TABLE 9: The replacement rates of 20 types of anomalies injections of SG.

Anomaly type	replaceRate
Transaction order dependence	0.60%
Results of contract execution affected by miners	0.38%
Unhandled false exception	0.54%
Integer overflow and underflow	0.38%
Use *tx.origin* for authentication	0.65%
Re-entrancy	0.80%
Wasteful contracts	0.40%
Short address attack	0.33%
Suicide contracts	0.46%
Locked ether	0.58%
Forced to receive ether	0.52%
Pre-sent ether	0.48%
Uninitialized local/state variables	0.60%
Hash collisions with multiple variable length arguments	0.20%
Specify *function* variable as any type	0.23%
Dos by complex *fallback* function	0.46%
Public function that could be declared *external*	0.36%
Non-public variables are accessed by *public*/*external*	0.54%
Nonstandard naming	0.65%
Unlimited compiler versions	0.44%

in the datasets. Liao et al. [25] defined a toy contract as a smart contract that has not been previously invoked, as genuine smart contracts are generally created for sending transactions. Therefore, to identify if a smart contract is more likely to be a toy contract, we check if the contract was previously invoked and the number of transactions sent to the contract is 0. We conducted a statistical analysis on the transactions received by the contracts of the dataset and found that 96.4% of contracts receive at least 1 transaction. This indicates that most of the contracts are less likely to be toy contracts. However, it is acknowledged that a contract that received a single transaction is at a very high risk of being a toy/education/test (i.e. irrelevant) contract. Second, *the numbers of labeled dataset and datasetθ are set to 50 and 1000* respectively due to the consideration of labor cost and time. From a macro perspective, 50 contacts per anomaly type $*$ 20 anomaly types indicate that we have to review 1000 contracts to assess the injection results of *SCAnoGenerator* and we also need to evaluate the injection results of *SolidiFI* injecting into 322 contracts. It can be seen that we need to manually review all the places where anomalies are injected, and more than one anomaly is injected into a contract. Therefore, after fully considering the time and labor costs, we chose 50 contracts as the final experimental object. 1000 contacts are selected since *SCAnoGenerator* needs to search for injectable contracts on a large scale and our evaluation intention is to measure the anomaly injection capacity of *SCAnoGenerator* in large contract sets. Moreover, *SCAnoGenerator* anomaly injection requires to run *solc* and *Slither*, which is relatively time-consuming. In order to eliminate this threat, we provide details of these datasets in Table 3 to prove the credibility. Third, the *tools that SCAnoGenerator relies on*. *SCAnoGenerator* relies on two open-source tools (i.e., *solc* and *Slither*) to construct a contract's control and data flows. The performance of these tools may also affect the validity of *SCAnoGenerator*. Nevertheless, *solc* and *Slither* are currently widely used open-source tools, which are regularly maintained by dedicated organizations and developers. This will reduce the impact of this threat.

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

The potential threat of **conclusion validity** comes from the *timeout threshold*. To keep the time-consuming evaluation within an acceptable range, if the time taken by an analysis tool exceeds our preset timeout threshold, we will terminate the execution of the tool and view it as an analysis failure. The setting of the timeout threshold might cause our evaluation results biased. To reduce the impact of this threat, we investigate the average running time of these tools based on the work in [27], [60] and set timeout threshold to 15 minutes. We have also increased the detection time of tools to 30 minutes. We found that, even if the time is increased by 15 minutes, there is no change in the results of static detection tools (i.e., *Slither* and *SmartCheck*) and dynamic tools (i.e., *Manticore* and *Maian*). For the remaining two detection tools (i.e., *Mythril* and *Securify*), when we adjusted the time to 30 minutes, *Mythril*'s captureRate on *Integer overflow and underflow*, *Re-entrancy* and *Specify function variable as any type* is not changed, and the captureRate on the remaining anomaly types is increased by 2.4% in total. Similarly, *Securify*'s captureRate on *Locked ether* and *Pre-sent ether* is not changed, and the captureRate of the remaining anomaly types is increased by 7.0% in total.

6 Related work

The section includes four parts: anomaly injection tools, smart contract analysis tools, smart contract datasets and exploiting smart contract anomalies.

6.1 Anomaly injection tools

Some researchers develop anomaly injection tools to build large-scale vulnerable program datasets. Bonett et al. propose *μSE* [61], a mutation-based Android static analysis tool evaluation framework. It systematically evaluates Android static analysis tools through mutation analysis to detect weaknesses of these tools. Their work validates the role of anomaly injection tools in finding weaknesses in analysis tools. Pewny et al. propose *EvilCoder* [62], an anomaly injection tool that automatically finds the locations of potentially vulnerable source code. It modifies the source code and outputs the actual vulnerabilities. *EvilCoder* first employs automated program analysis technologies to find functions for anomaly injection. And then, it conducts possible attacks by inserting statements or invalidating security measures. Our work is inspired by *EvilCoder*. Dolan-Gavitt et al. propose *LAVA* [63], an anomaly injection tool based on dynamic taint analysis. *LAVA* can quickly inject a large number of anomalies into programs to build a large-scale corpus of vulnerable programs. In addition, *LAVA* can also provide an input for each injected anomaly to trigger the anomaly. Ghaleb et al. propose *SolidiFI* [33], which is the first anomaly injection tool for Ethereum smart contracts. *SolidiFI* injects anomalies into contracts by injecting code snippets containing anomalies into all possible locations in the contracts. They employ *SolidiFI* to inject 9,369 anomalies of 7 types into 50 contracts. These contracts are then used to evaluate 6 analysis tools. The evaluation results show that these tools cause a large number of false positives and false negatives. Hajdu et al. [32] use software-implemented anomaly injection to evaluate the behavior of anomalistic smart contracts, and analyze the effectiveness of formal verification and runtime protection mechanisms in detecting injected anomalies. However, these work except for [33] cannot inject anomalies into Ethereum smart contracts, or does not provide useable open-source tools.

6.2 Evaluating smart contract analysis tools

Some studies are devoted to evaluating the detection abilities of smart contract analysis tools. Zhang et al. [45] propose an Ethereum smart contract anomaly 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. [64] 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. [27] 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 *SmartAnomalys* [65], an execution framework that integrates 10 analysis tools. Parizi et al. [59] evaluate 4 analysis tools using several handwritten datasets. They think that *SmartCheck* can detect the most anomalies, and *Mythril* has the highest accuracy. The existing evaluations of Ethereum smart contract analysis tools rely on either small-scale labelled handwritten datasets or unlabelled *real-world contract* datasets. This makes it impossible to effectively and precisely evaluate the real performance of analysis tools on anomaly detection. The emergence of *SCAnoGenerator* is expected to solve this dilemma.

6.3 Ethereum anomalistic smart contract datasets

Some organizations and researchers provide anomalistic Ethereum smart contract datasets to show developers examples of various anomalies and provide benchmarks for smart contract analysis tool evaluation. *SmartContractSecurity* [29] provides a list of 36 types of Ethereum smart contract anomalies and creates exemplary anomalistic contracts for each type of anomalies. *Crytic* [30] provides an anomalistic 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. [45] provide an anomalistic contract dataset covering 49 types of smart contract anomalies. This dataset is currently the largest handwritten dataset in terms of the number (173) of contracts. Durieux et al. [27] create two datasets. One contains 47,398 unlabeled *real-world contracts*. The other comprises 69 labeled anomalistic *handwritten contracts*. According to the smart contract anomaly classification scheme provided by *DASP* [66], they classify the anomalies in 69 contracts into 10 types. The labeled anomalistic contract datasets provided by the above work all share the following limitations: inadequate number of contracts, small contract code size, and lack of real business logic in contracts.

6.4 Exploiting smart contract anomalies

Some studies are performed to find and exploit anomalies in smart contracts. Jiang et al. [8] propose *ContractFuzzer*, a smart contract fuzzing tool. *ContractFuzzer* generates fuzzing inputs and uses these inputs to run smart contracts to trigger anomalies in smart contracts. Zhang et al. propose *ETPLOIT* [67], a fuzzing-based smart contract anomaly exploit generator. *ETHPLOIT* employs static taint analysis to generate transaction sequences that can trigger anomalies. It adopts a dynamic seed strategy to overcome hard constraints in the execution paths. Feng et al. propose *SMARTSCOPY* [68], an automatic generation tool for adversarial contracts. *SMARTSCOPY* adopts techniques such as symbolic execution to identify and exploit anomalies in the victim's smart contracts. Krupp et al. [69] provide the definitions of vulnerable contracts. They present *TEETHER*, which can generate exploits by analyzing the bytecode of the contracts.

7 Conclusion and future work

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

For future work, first, we plan to study how to improve *SCAnoGenerator* into a flexible and extensible anomaly injection framework. i.e., users only need to define some rules, and then *SCAnoGenerator* just can inject a new type of anomalies. Additionally, we will try to expand *SCAnoGenerator* to enable it to inject more types of anomalies.

While the manual inspection provides essential expert validation for complex anomaly types, we are actively exploring automated validation methods that could complement the manual process. This would help enhance both scalability and reproducibility in future iterations of our evaluation framework, allowing for a more streamlined and objective validation of injected anomalies without compromising accuracy.

ACKNOWLEDGMENTS

The work is supported by the National Natural Science Foundation of China (No.62272145 and No.U21B2016), Hong Kong ITF Project (No. GHP/052/19SZ), Hong Kong RGC Projects (No. PolyU15222320, PolyU15224121), Digital Finance CRC (DFCRC) Industrial PhD Scholarship, and CloudTech-RMIT Green Bitcoin Joint Research Program. Special thanks to master student Feng Xiao for his contribution to *SCAnoGenerator*, as he did extensive research at the beginning of the work.

REFERENCES

[1] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, 2009.
[2] W. Zou, D. Lo, P. S. Kochhar, X. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart contract development: Challenges and opportunities," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.
[3] V. Buterin *et al.*, "Ethereum white paper," *GitHub repository*, pp. 22–23, 2013.
[4] C. Natoli and V. Gramoli, "The blockchain anomaly," in *2016 IEEE 15th international symposium on network computing and applications (NCA)*. IEEE, 2016, pp. 310–317.
[5] C. Diligence. Ethereum smart contract security best practices. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/
[6] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1503–1520.
[7] I. Grishchenko, M. Maffei, and C. Schneidewind, "Ethertrust: Sound static analysis of ethereum bytecode," *Technische Universität Wien, Tech. Rep*, 2018.
[8] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 259–269.
[9] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang, "GasChecker: Scalable analysis for discovering gas-inefficient smart contracts," *IEEE Transactions on Emerging Topics in Computing*, 2020.
[10] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang, "Towards saving money in using smart contracts," in *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. IEEE, 2018, pp. 81–84.
[11] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 442–446.
[12] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 653–663.
[13] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." in *NDSS*, 2018.
[14] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 65–68.
[15] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.
[16] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2018, pp. 9–16.
[17] C. F. Torres, J. Schütte *et al.*, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 664–676.
[18] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016, pp. 254–269.
[19] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 67–82.
[20] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 778–788.
[21] J. He, M. Balunovi, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *the 2019 ACM SIGSAC Conference*, 2019.
[22] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1661–1677.
[23] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network." in *IJCAI*, 2020, pp. 3283–3290.
[24] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai, "Empirical evaluation of smart contract testing: what is the best choice?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 566–579.
[25] Z. Liao, S. Song, H. Zhu, X. Luo, Z. He, R. Jiang, T. Chen, J. Chen, T. Zhang, and X. Zhang, "Large-scale empirical study of inline assembly on 7.6 million ethereum smart contracts," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 777–801, 2022.
[26] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang, "Towards saving money in using smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, 2018, pp. 81–84.
[27] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 530–541.
[28] SMARX. (2021, Mar.) Warmup. [Online]. Available: https://capturetheether.com/challenges/
[29] SmartContractSecurity. (2021, Jan.) Smart contract weakness classification and test cases. [Online]. Available: https://swcregistry.io/
[30] T. of Bits. (2021, Jan.) Examples of solidity security issues. [Online]. Available: https://github.com/crytic/not-so-smart-contracts
[31] OpenZeppelin. (2021, Jan.) Web3/solidity based wargame. [Online]. Available: https://ethernaut.openzeppelin.com/
[32] Á. Hajdu, N. Ivaki, I. Kocsis, A. Klenik, L. Gönczy, N. Laranjeiro, H. Madeira, and A. Pataricza, "Using fault injection to assess blockchain systems in presence of faulty smart contracts," *IEEE Access*, vol. 8, pp. 190 760–190 783, 2020.
[33] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 415–427. [Online]. Available: https://doi.org/10.1145/3395363.3397385
[34] smartdec. (2020, Apr.) a static analysis tool that detects vulnerabilities and bugs in solidity programs. [Online]. Available: https://tool.smartdec.net/
[35] crytic. (2020, Apr.) Static analyzer for solidity. [Online]. Available: https://github.com/crytic/slither
[36] ConsenSys. (2020, Jan.) Security analysis tool for evm bytecode. [Online]. Available: https://github.com/ConsenSys/mythril
[37] trailofbits. (2020, Apr.) Symbolic execution tool. [Online]. Available: https://github.com/trailofbits/manticore

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

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

[40] G. A. Oliva, A. E. Hassan, and Z. M. J. Jiang, "An exploratory study of smart contracts in the ethereum blockchain platform," *Empirical Software Engineering*, vol. 25, no. 3, pp. 1864–1904, 2020.

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

[42] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, pp. 1–36, 2017.

[43] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2021.

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

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

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

[47] B. Li, Z. Pan, and T. Hu, "Redefender: Detecting reentrancy vulnerabilities in smart contracts automatically," *IEEE Transactions on Reliability*, 2022.

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

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

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

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

[52] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 317–331.

[53] D. Hedley and M. A. Hennell, "The causes and effects of infeasible paths in computer programs," in *Proceedings of the 8th international conference on Software engineering*, 1985, pp. 259–266.

[54] R. Bodik, R. Gupta, and M. L. Soffa, "Refining data flow information using infeasible paths," in *Software Engineering—ESEC/FSE'97*. Springer, 1997, pp. 361–377.

[55] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defectchecker: Automated smart contract defect detection by analyzing evm bytecode," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

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

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

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

[59] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," in *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '18. USA: IBM Corp., 2018, p. 103–113.

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

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

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

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

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

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

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

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

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

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

Pengcheng Zhang received the Ph.D. degree in computer science from Southeast University in 2010. He is currently an associate professor in College of Computer Science and Software Engineering, Hohai University, Nanjing, China. His research interests include software engineering, service computing and data mining. He has published in premiere or famous computer science journals. He is a memeber of the IEEE.

Ben Wang received the M.S. degree in computer science and technology from Hohai University in 2023. He is currently working towards the Ph.D. degree with the College of Computer Science and Software Engineering, Hohai University, Nanjing, China. His current research interests include smart contract security and software engineering.

Xiapu Luo is an assistant professor with the Department of Computing and an Associate Researcher with the Shenzhen Research Institute, The Hong Kong Polytechnic University. He received the Ph.D. degree in Computer Science from The Hong Kong Polytechnic University, and was a Post-Doctoral Research Fellow with the Georgia Institute of Technology. His research focuses on smartphone security and privacy, network security and privacy, and Internet measurement.

Hai Dong is a senior lecturer at the School of Computing Technologies, RMIT University, Melbourne, Australia. He received a PhD from Curtin University, Perth, Australia. His research interests include Service-Oriented Computing, Edge Computing, Blockchain, Cyber Security, Machine Learning, and Data Science. He is a senior member of the IEEE.