A Framework and Data Set for Bugs in Ethereum Smart Contracts

***Abstract*—Ethereum is the largest blockchain platform that supports smart contracts. Users deploy smart contracts by publishing the smart contract's bytecode to the blockchain. Since the data in the blockchain cannot be modified, even if these contracts contain bugs, it is not possible to patch deployed smart contracts with code updates. Moreover, there is currently neither a comprehensive classification framework for Ethereum smart contract bugs, nor detailed criteria for detecting bugs in smart contracts, making it difficult for developers to fully understand the negative effects of bugs and design new approaches to detect bugs. In this paper, to fill the gap, we first collect as many smart contract bugs as possible from multiple sources and divide these bugs into 9 categories by cutting and expanding the *IEEE Standard Classification for Software Anomalies*. Then, we design the criteria for detecting each kind of bugs, and construct a dataset of smart contracts covering all kinds of bugs. With our framework and dataset, developers can learn smart contract bugs and develop new tools to detect and locate bugs in smart contracts. Moreover, we evaluate the state-of-the-art tools for smart contract analysis with our dataset and obtain some interesting findings: 1) *Mythril*, *Slither* and *Remix* are the most worthwhile combination of analysis tools. 2) There are still 12 kinds of bugs that cannot be detected by any analysis tool.**

***Index Terms*—Ethereum, Solidity, Smart contract bug**

I. Introduction

Millions of smart contracts have been deployed onto Ethereum, the largest blockchain that supports smart contracts. They are typically developed with high-level programming languages and then compiled into bytecode, which will be deployed to the blockchain through transactions. Note that the deployed bytecode cannot be modified for patching the bugs. Unfortunately, similar to traditional computer programs, it is difficult to avoid bugs in smart contracts. Recent years have witnessed various bugs in smart contracts, resulting in huge losses. eg., the *re-entrancy bug* [1] in *the DAO* smart contract [2] led to a loss of $60 million.

Although recent studies proposed a number of tools [3]–[11] for detecting the bugs in smart contracts, Ye et al. [12] found that they can only discover some kinds of known smart contract bugs. One possible reason for this situation is the lack of a comprehensive collection and classification of all existing smart contract bugs so that existing tools just aim at portions of bugs. At the same time, recent studies reveal that one major reason for the prevalence of smart contract bugs is the lack of a comprehensive classification framework for smart contract bugs [1]. Although a few studies summarized and classified some kinds of bugs in smart contracts [1], [13]–[15], they have the following limitations:

- *Existing studies do not cover the known bugs in smart contracts.* Dingman et al. [13] considered 49 kinds of bugs and classified them using *NIST* framework. However, they grouped only 24 kinds of bugs into well-defined categories, and put the remaining 25 kinds of bugs into *other* category without further classification. Moreover, they also included the bugs that have been fixed. Smartdec [16] divided smart contract bugs into three levels: blockchain, language and model, and then classified the bugs at each level. Chen et al. [14] divided 20 kinds of bugs into 5 categories. Durieux et al. [15] divided bugs into 10 categories. However, these studies have the following limitation: they only list part of bug kinds.
- *Lack of detailed bug detection criteria*. Existing work [1], [13]–[15] only describes the causes of various kinds of bugs, but does not give the criteria for detecting the existence of bugs, making it difficult for developers to design algorithms and tools to detect bugs. eg., existing tools cannot detect the *short address attack bug* due to the lack of criteria for the existence of *short address attack bug*.
- *The existing datasets for smart contract bugs are incomplete.* eg., *SmartContractSecurity* [17] includes 33 kinds of bugs, but only provides sample smart contracts for 31 kinds of bugs; *crytic* [18] provides sample smart contracts for *Solidity* security issues, but only covers 12 kinds of bugs; Durieux et al. [15] provide a dataset containing 69 problematic smart contracts, but they only covered 10 kinds of bugs. Without a dataset that covers all kinds of bugs and includes the corresponding vulnerable smart contracts, it is difficult to comprehensively evaluate the performance of existing tools for finding bugs in smart contracts.

In this paper, to fill in the gap, we first carefully collect known bugs of Ethereum smart contracts from many sources, including, academic literature, the Web, blogs, and related open-source projects, and finally obtain 323 records describing Ethereum smart contract bugs. Then, by reviewing the *Ethereum Wiki* [19], *Ethereum Improvement Proposals* [20] and the development documents of *Solidity* [21], we remove the bugs that had been fixed by Ethereum. We also merge the bugs caused by the same behavior, and eventually 49 kinds of bugs left. After that, by cutting and expanding the *IEEE Standard Classification for Software Anomalies*, we classify 49 kinds of bugs into 9 categories based on the causes of these bugs, and give the detect criteria for each kind of bug. Finally, according to the classification framework, we provide

a smart contract data set containing the collected bugs and evaluate several smart contract analysis tools using this data set. We call the framework and dataset as *Jiuzhou*, which can be found at https://github.com/xf97/JiuZhou.

In summary, we make the following contributions:

- We propose a comprehensive framework for the bugs in Ethereum smart contracts based on *IEEE Standard Classification for Software Anomalies*. We collect these bugs from many sources and classify them into 9 categories.
- We give the detect criteria for each kind of bug. According to the cause of each bug, the most common form of bugs, and the false positives and omissions generated by various analysis tools when detecting these bugs, we give the detect criteria for each bug. The detect criteria are based on features, that is, if there are certain features in the contract, then there is some kind of bug. In the future, we will study how to formally define detect criteria.
- We provide a data set of problematic smart contracts. The smart contracts in the data set cover all kinds of bugs. It contains 176 smart contracts, including contracts that contain bugs, contracts that fix bugs and induced contracts. By reading the smart contracts in the data set, developers and researchers can understand the programming patterns that are likely to cause bugs and the solutions to avoid bugs.
- We use the data set as a benchmark to evaluate various smart contract analysis tools. Based on our evaluation results, we obtain the detection abilities of nine smart contract analysis tools and find that there are still 12 kinds of bugs that cannot be detected by any of the analysis tools evaluated, and our evaluation also has some interesting findings.

The rest of this paper is organized as follows: Section 2 introduces the necessary background. Section 3 presents the Ethereum smart contract bug classification framework, describes the characteristics and detect criteria of each bug, and gives the severity level of each bug. Section 4 introduces the data set that matches the bug classification. Section 5 uses the *Jiuzhou* data set to evaluate smart contract analysis tools. Section 6 describes the related work. Finally, Section 7 concludes the paper and proposes future work.

II. BACKGROUND

A. Smart contract

When the conditions specified in the contract are met or the smart contracts are called, smart contracts can be executed automatically on blockchain [22]. In Ethereum, each smart contract or user is assigned a unique address. Smart contracts can be invoked by sending transactions to the address of the contract. *Ether* is the cryptocurrency used by Ethereum, and both contracts and users can trade *ethers*. To avoid abusing the computational resources, Ethereum charges *gas* from each executed smart contract statement.

B. Solidity

Solidity is the most widely used programming language for developing Ethereum smart contracts [21], and also is a Turing-complete and high-level programming language capable of expressing arbitrarily complex logic. Before deployment, the smart contracts written by *Solidity* are compiled into bytecode of Ethereum virtual machine. *Solidity* provides many built-in symbols to perform various functions of Ethereum. eg., *transfer* and *send* are used to perform the transfer of ethers, and keywords such as *require* and *assert* are used to handle errors. *Solidity* is a fast-evolving language. The same keyword may have different semantics in different language versions. In general, when using *Solidity* to develop smart contracts, developers can specify the *Solidity* version used by the contracts.

C. IEEE Standard Classification for Software Anomalies

The *IEEE Standard Classification for Software Anomalies* [23] provides a unified method for the classification of traditional software anomalies. In the standard, *error*, *fault*, *defect*, *problem*, and *bug* are uniformly described as *anomalies*. In its latest version, software anomalies are classified into six categories: *data*, *interface*, *logic*, *description*, *syntax*, *standard*. The standard also provides ranking criteria for the effect and priority of software anomalies. Researchers can flexibly cut or extend this standard to adapt to different types of software.

III. A CLASSIFICATION FRAMEWORK FOR SMART CONTRACT BUGS

To build a comprehensive classification framework, we collect smart contract bugs from many sources, including academic literature, the Web, blogs, and related open-source projects. Since there is no uniform bug naming criterion, the same bug may have different names. Consequently, we first merge bugs according to their behaviors. Then, according to the causes of all bugs, we divided all bugs into 9 categories. Each category contains several sub-categories, and the sub-categories contain several kinds of bugs. Finally, according to the effect of different bugs, we give each bug a severity rating.

A. Collect smart contract bugs

First, we collect smart contract bugs from academic literature, the Web, blogs, and other resources. For academic literature, we use *smart contract vulnerabilities*, *smart contract bugs*, *smart contract defects*, *smart contract problems*, and *smart contract anomalies* as search keywords to search for papers published since 2014 in *ACM digital library* [24] and *IEEE Xplore digital library* [25]. The reason for the paper after 2014 was chosen is that Ethereum started ICO (initial coin offering) in 2014. For the Web and blogs, we mainly focus on the *Github homepage of Ethereum* [26], *the development documents of Solidity* [21], *the official blogs of Ethereum* [27], *the Gitter chat room* [28], *Ethereum Improvement Proposals* [20] and other resources. Second, related open-source projects are also our focus since the open-source community plays an important role in the field of software security [29].

Specifically, we use *smart contract bugs*, *smart contract problems*, *smart contract defects*, *smart contract vulnerabilities*, and *smart contract anomalies* as search keywords to retrieve related open-source projects on *GitHub* [30]. Besides, many smart contract analysis tool projects are also open-sourced on *GitHub* [30], and some of the project documents describe information about smart contract bugs. Therefore, we also use *smart contract analysis tools* and *smart contract security* as search keywords to retrieve open-source projects on *Github*. We focus on the projects for Ethereum smart contracts. After removing duplicate search results, we obtain a total of 266 projects. Third, many famous Ethereum smart contract analysis tools can detect smart contract bugs. We send emails to the authors of these tools asking what kinds of bugs they detect. We also look at the kinds of bugs detected by the *Solidity static analysis* function of *Remix* [31]. Finally, from the resources mentioned above, we collected 323 records describing Ethereum smart contract bugs.

To continuously collect bugs, we develop a program called *BugGetter*[1]. *BugGetter* runs regularly (now set to 15 days, adjustable), and sends query requests to *Github* every time it runs. *BugGetter* uses keywords such as *smart contract vulnerabilities*, *smart contract bugs*, *smart contract defects*, *smart contract problems*, *smart contract security*, and *smart contract analysis tools* to construct query requests to *Github*, and parses out the list of projects and the update time of these projects. By comparing previously obtained projects and their update time, *BugGetter* will send us an email if a new project appears or an existing project is updated. After receiving the email, we will manually check all changes and update the collected bug results in time.

B. Merge smart contract bugs

Because there is no uniform bug naming criterion, even if the names of the collected bugs are different, some bugs may point to the same bug. Consequently, we need to merge the duplicate bugs. The collected bugs generally have two attributes, namely, the behaviors causing the bug and the consequences caused by the bug. If there is a bug *A*. Let,

- the behaviors causing bug *A* be *b(A)*,
- the consequences caused by bug *A* be *c(A)*.

If there are two bugs, *A* and *B*. Then *A* and *B* are merged according to the following steps: 1) $b(A) \neq b(B)$, *A* and *B* are not merged. 2) $b(A) = b(B), c(A) \neq c(B)$, in this case, *c(A)* and *c(B)* respectively cover part of the consequences of the bug. We merge *A* and *B*, rename the merged bug, summarize *c(A)* and *c(B)*, and give the consequences after they are merged. 3) $b(A) = b(B), c(A) = c(B)$, in this case, we choose the name that better reflects the characteristics of the bug as the name of the merged bug, and then *A* and *B* are merged.

After the duplicate bugs are merged, we verify the validity of each bug (that is, the bug has not been fixed), and delete the fixed bugs. Finally, 49 kinds of bugs are left. We list the correspondence of bugs before and after the merger via the *url*[2], which allows us to trace the merge process.

C. Classify smart contract bugs

Classification criteria and results

According to *IEEE Standard Classification for Software Anomalies [23]* issued in 2010, software anomalies are classified into six categories: *data*, *interface*, *logic*, *description*, *syntax*, *standard*. Among them, we do not consider the *syntax* category, because a smart contract with syntax bugs cannot be compiled into bytecode and cannot be deployed in Ethereum. Besides, the bugs caused by *gas*, *lack of privacy on the blockchain*, *smart contract authority control*, *smart contract interactions*, and *smart contract support software* are Ethereum-specific software anomalies. Consequently, the original classification provided by *IEEE Standard Classification for Software Anomalies [23]* cannot accurately classify these bugs. To accurately classify all kinds of bugs in smart contracts, we add four new categories: *security (for lack of privacy and authority control)*, *performance (for gas consumption)*, *interaction (for smart contract interaction and ethers exchange)*, and *environment (for smart contract support software)*. Therefore, we divide smart contract bugs into the following nine categories lexicographically:

1) ***Data***. Bugs in data definition, initialization, mapping, access, or use, as found in a model, specification, or implementation.
2) ***Description***. Bugs in the description of the software or its use, installation, or operation.
3) ***Environment***. Bugs due to errors in the supporting software.
4) ***Interaction***. Bugs that cause by interaction with other Ethereum addresses.
5) ***Interface***. Bugs in specification or implementation of an interface.
6) ***Logic***. Bugs in decision logic, branching, sequencing, or a computational algorithm, as found in natural language specifications or implementation language.
7) ***Performance***. Bugs that cause increased *gas* consumption.
8) ***Security***. Bugs that threaten contract security, such as authentication, privacy/confidentiality, property.
9) ***Standard***. Nonconformity with a defined standard.

During the process of merging bugs, by checking *Ethereum Improvement Proposals* [20], *the Ethereum Wiki* [19], and the development documents of *Solidity* [21], we removed bugs that have been fixed by Ethereum (eg., the *call depth attack*, which was fixed in the *EIP150* [32]). Some kinds of bugs are caused by specific *Solidity* versions. Because it is still possible to use these versions of *Solidity* to develop smart contracts, we list the range of *Solidity* versions that cause these kinds of bugs. The remaining kinds of bugs exist in any version of *Solidity*.

We divide 49 kinds of bugs into 9 categories, each category is divided into several sub-categories, and each

[1]https://github.com/xf97/BugGetter

[2]https://github.com/xf97/JiuZhou/blob/master/Correspondence.xlsx

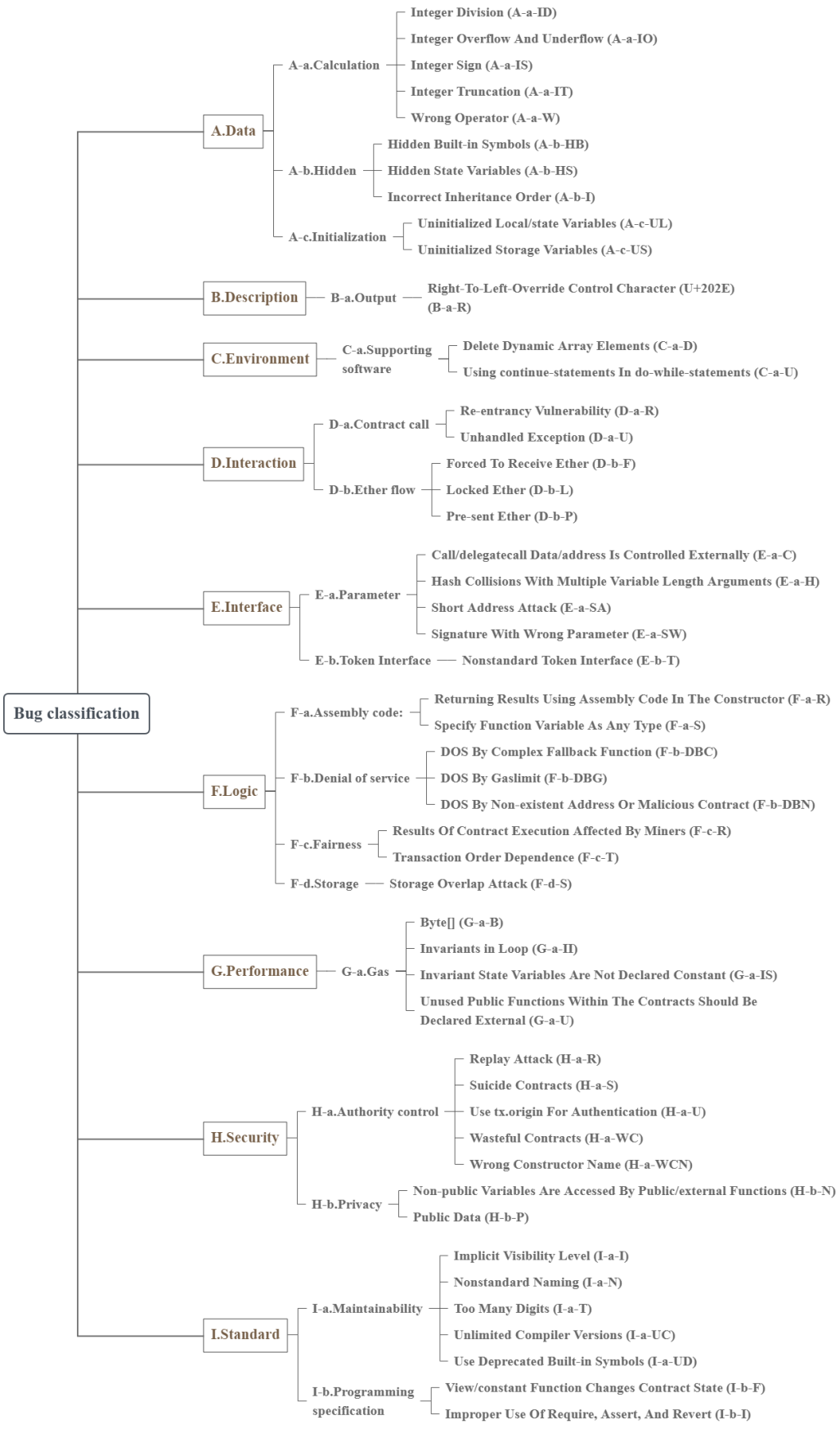

Fig. 1. Smart contract bug statistics and classification

sub-category contains several kinds of bugs. Fig 1 shows the classification results of 49 kinds of bugs. For ease of expression, we assign a corresponding abbreviated name for each bug. The composition rule of the abbreviated name is: *category number-subcategory number-short name*. The *short name* is an acronym that can be distinguished from the *short names* of other kinds of bugs in the same sub-category. Classifying bugs based on their descriptions is a manual process based on natural language descriptions, so this may introduce our subjectivity and ambiguity. To reduce the impact of this problem, three researchers who are familiar with smart contract bugs participated in the classification process and reached a consensus through discussion. Due to space limitations, in this paper, we only introduce some kinds of bugs in detail. For the full version, please access this *url*[3].

Some kinds of smart contract bugs

In this part, we will introduce the causes, consequences, and detect criteria of seven kinds of bugs in detail, and illustrate these bugs with examples. These kinds of bugs are either bugs that are difficult to understand the attack process or bugs not mentioned in other work [1], [13], [14]. Table I lists these 7 kinds of bugs.

TABLE I
7 KINDS OF BUGS INTRODUCED IN DETAIL

Bug name abbreviation	A-a-IS, A-a-W, A-c-US, D-a-R E-a-SA, E-a-SW, F-c-T

Integer Sign (A-a-IS) [5]:

- *Cause*: In *Solidity*, Converting *int* type to *uint* type (and vice versa) may produce incorrect results.
- *Consequence*: This kind of bugs may result in incorrect integer operation results, which will affect the function of the contract. When the wrong result is used to indicate the number of ethers (or tokens), this kind of bugs will cause economic losses.
- *Example*: Consider the smart contract in Fig 2, if the attacker calls the *withdrawOnce* function and specifies that the value of *amount* is a negative number, then this call will pass the check and transfer out the ether that exceeds the limit (1 ether).
- *Detect criteria*: This kind of bugs exists when the contract contains the following features: 1) Forcibly convert an *int* variable to a *uint* variable. 2) The contract does not check whether this *int* variable is negative.

Wrong Operator (A-a-W) [17]:

- *Cause*: Users can use =+ and =- operators in the integer operation without compiling errors (up to and including version 0.4.26).
- *Consequence*: Consistent with the consequences of *integer sign* bug.
- *Example*: Consider the smart contract in Fig 3, the user can adjust the value of *myNum* by calling *addOne*/*subOne* function. When *myNum* and *WinNum* are equal, the user gets all the ethers of the contract. But because the developers write the wrong operators (Line 15, 19), the value of *myNum* will not change.
- *Detect criteria*: This kind of bugs exists when the following feature exists in the contract: 1) There is a =+ or =- operator in the contract.

[3]https://github.com/xf97/JiuZhou/blob/master/Jiuzhou_Full_version.pdf

```
1  pragma solidity 0.6.2;
2  contract signednessError{
3      mapping(address => bool) public transferred;
4      address public owner;
5      constructor() public payable{
6          owner = msg.sender;
7          require(msg.value > 0 && msg.value % 1 ether == 0);
8      }
9      function withdrawOnce (int amount) public {
10         if ( amount > 1 ether || transferred [msg.sender]) {
11             revert() ;
12         }
13         msg.sender.transfer(uint(amount));
14         transferred [msg.sender] = true ;
15     }
16 }
```

Fig. 2. A contract that contains *integer sign* bug

```
1  pragma solidity 0.4.26;
2  contract wrongOpe{
3      uint256 private myNum;
4      address public owner;
5      uint256 public winNum;
6      uint256 public constant OpeNum = 1;
7      constructor(uint256 _num, uint256 _win) public payable{
8          myNum = _num;
9          winNum = _win;
10         owner = msg.sender;
11         require(msg.value == 1 ether);
12     }
13     function addOne() external payable{
14         require(msg.value == 1 wei);
15         myNum =+ OpeNum; //wrong operator
16         isWin(); }
17     function subOne() external payable{
18         require(msg.value == 1 wei);
19         myNum =- OpeNum; //wrong operator
20         isWin(); }
21     function isWin() internal{
22         if(myNum == winNum)
23             msg.sender.transfer(address(this).balance); }
24 }
```

Fig. 3. A contract contains a *wrong operator* bug

Uninitialized Storage Variables (A-c-US) [17]:

- *Cause*: The uninitialized storage variable serves as a reference to the first state variable, which may cause the state variable to be inadvertently modified (up to and including version 0.4.26).
- *Consequence*: This kind of bugs may cause key state variables to be rewritten inadvertently, and eventually, the function of the contract will be affected.
- *Example*: Consider the smart contract in Fig 4, when the user calls the function *func*, the *owner* will be re-assigned to 0x0.
- *Detect criteria*: This kind of bugs exists when the contract contains the following features: 1) the developers do not initialize the storage variables in the contract.

Re-entrancy Vulnerability (D-a-R) [11], [33]:

- *Cause*: When the *call*-statement is used to call other contracts, the callee can call back the caller and enter the caller again.
- *Consequence*: This kind of bugs is one of the most dangerous smart contract bugs, which will cause the

```
1  pragma solidity 0.4.26
2  contract Uninitialized{
3      address owner = msg.sender;
4      struct St{
5          uint a;
6      }
7      function func() {
8          St st;
9          st.a = 0x0; //owner is override to 0.
10     }
11 }
```

Fig. 4. A contract contains a *uninitialized storage variables* bug

contract balance (ethers) to be stolen by attackers.

- *Example*: We use an example to illustrate the *re-entrancy vulnerability*. In Fig 5, the contract *Re* is a contract with a *re-entrancy vulnerability*, and the *balance* variable is a map used to record the correspondence between the address and the number of tokens. The attacker deploys the contract *Attack*, and the value of the parameter *_reAddr* is set to the address of the contract *Re*. In this way, the *re* variable becomes an instance of the contract *Re*. Then,
  - *Step 1*: The attacker calls the *attack* function to deposit ethers into the contract *Re* and then calls the *Re.withdraw* function to retrieve the deposited ethers.
  - *Step 2*: The contract *Re* executes the *withdraw* function and uses a *call*-statement to send ethers to the contract *Attack*. At this time, the power of control is transferred to the contract *Attack*, and the contract *Attack* responds to the transfer using the *Attack.fallback* function.
  - *Step 3*: The *Attack.fallback* function calls the *Re.withdraw* function to withdraw the ethers again. Therefore, the statement (*deduct-statement*) deducting the number of tokens held by the contract *Attack* will not be executed.
- *Detect criteria*: When there are the following features in the contract, it will cause the *reentrancy bug*: 1) The *call-statement* is used to send ethers. 2) The amount of *gas* to be carried is not specified. 3) No callee's response function is specified. 4) Ethers are transferred first and callee's balance is deduced later.

Short Address Attack (E-a-SA) [34]:

- *Cause*: 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.
- *Consequence*: This kind of bug may cause the attacker to manipulate tokens (ethers) equivalent to several times the number he requested.
- *Example*: We use an example to illustrate *short address attacks*.
  - *Step 1*: *Tom* deploys token contract *A* on Ethereum, which contains the *sendCoin* function. The code of *sendCoin* is shown in Fig 6.
  - *Step 2*: *Jack* buys 100 tokens of contract *A*, then registers for an Ethereum account with the last two digits zero (eg. 0x1234567890123456789012345678901234567800).
  - *Step 3*: *Jack* calls the function *sendCoin* with the given parameters, *_to*: 0x123456789012345678901234567890123456 78 (missing last two digits 0), *_amount*: 50.
  - *Step 4*: The value of *_amount* is less than 100, so it passes the check. However, because the bits of *_to* is insufficient, the first two bits (0) of *_amount* will be added to the *_to* when the transaction data is packed. Therefore, to make up for the length of *_amount*, the Ethereum virtual machine will add 0 to the last two bits. In the end, the value of *_amount* is expanded by four times.
- *Detect criteria*: When there are the following features in the contract, it will cause the *short address attack*: 1) The contract uses a function to transfer ethers or tokens. 2) The number of tokens (ethers) and the address for receiving tokens (ethers) are provided by external users. 3) There is no operation to check the length of the received tokens (ethers) address in the function.

Signature With Wrong Parameter (E-a-SW) [35]:

- *Cause*: When the parameters of the *ecrecover()* are wrong, the *ecrecover()* will return 0x0.
- *Consequence*: This kind of bugs will allow the attacker to pass the authentication and then the attacker can manipulate the token (ethers) held by the *0x0* address.
- *Example*: Considering the smart contract in Fig 7, when the attacker gives the wrong parameters (*v*, *r*, *s*) and the value of the specified parameter *_id* is *0x0*, the attacker can pass the identity verification (Line 10), which eventually leads to the ethers in the contract are destroyed.
- *Detect criteria*: This kind of bugs exists when the contract contains the following features: 1) There is an operation in the contract that uses the *ecrecover()* to calculate the public key address. 2) The contract does not deal with the case where the *ecrecover()* returns *0x0*.

Transaction Order Dependence (F-c-T) [36]:

- *Cause*: Miners can decide which transactions are packaged into the blocks and the order in which transactions are packaged. The current main impact of this kind of bugs is the *approve* function in the *ERC20* token standard.
- *Consequence*: This kind of bugs will enable miners to influence the results of transaction execution. If the results of the previous transactions will 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.
- *Example*: The *approve* function allows one address to approve another address to spend tokens on his behalf. The standard implementation of the *approve* function is shown in Fig 8. We assume that *Alice* and *Tom* are two Ethereum users, and *Tom* runs an Ethereum node. The following steps reveal how *Tom* uses the *transaction order dependence* bug to monetize:

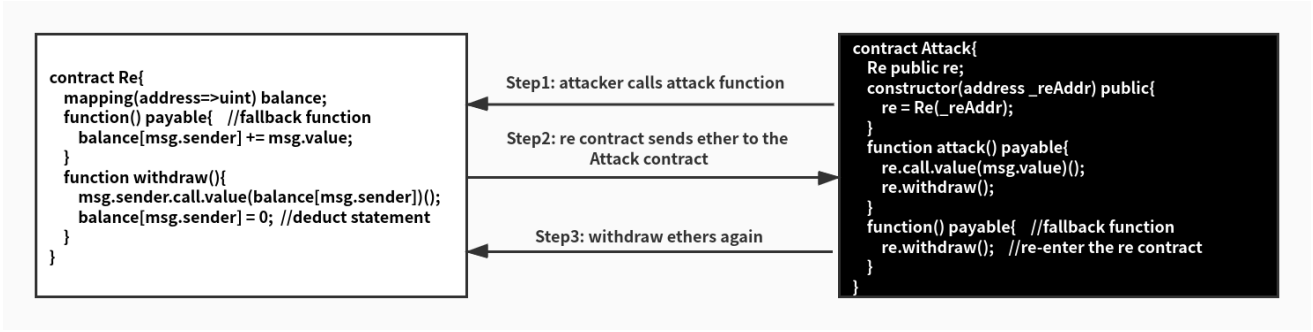

Fig. 5. An example of *re-entrancy vulnerability*

```
function sendCoin(address _to, uint256 _amount) returns(bool){
    if(balance[msg.sender] < _amount)
        return false;
    //using safemath library for uint256
    balance[msg.sender] = balance[msg.sender].sub(_amount);
    balance[_to] = balance[_to].add(_amount);
    Transfer(msg.sender, _to, _amount);
    return true;
}
```

Fig. 6. Objective function of *short address attack*

```
pragma solidity 0.6.2;
contract SigWrongPara{
    bytes32 private idHash;
    constructor() public payable{
        require(msg.value > 0);
        idHash = keccak256(abi.encode(msg.sender));
    }
    function getMyMoney(address payable _id,  uint8 v,
      bytes32 r, bytes32 s) external returns(bool){
        if(_id != ecrecover(idHash, v, r, s))
            return false;
        _id.transfer(address(this).balance);
        return true;
    }
}
```

Fig. 7. A contract contains a *signature with wrong parameter* bug

  - *Step 1*: Assume *Alice* has approved *Tom* to spend n of the tokens she holds. Now *Alice* decides to change *Tom*'s quota to m tokens, so *Alice* sends a transaction to modify *Tom*'s quota.
  - *Step 2*: Since *Tom* runs an Ethereum node, he knows that *Alice* will change his quota to m tokens. Then, *Tom* sends a transaction (e.g., Using the *transferFrom* function of the *ERC20* token standard to transfer n tokens to himself) to spend *Alice*'s n tokens, and pays a lot of *gas* to make his transaction executed first.
  - *Step 3*: The node that obtains the accounting right packs transactions. Because *Tom* pays more *gas*, *Tom*'s transaction will be executed before *Alice*'s transaction. Therefore, *Tom* spent n tokens of *Alice* first and then is granted a quota of m tokens by *Alice*, which caused *Alice* to suffer losses.

- *Detect criteria*: This kind of bugs exists when the contract contains the following features: 1) The contract contains the *approve* function of the *ERC20* token standard. 2) In the *approve* function, the quota of the approved address is set from one nonzero value to another nonzero value.

```
function approve(address spender, uint256 value)
public returns (bool) {
    require(spender != address(0));
    _allowed[msg.sender][spender] = value;
    emit Approval(msg.sender, spender, value);
    return true;
}
```

Fig. 8. The standard implementation of the *approve* function in the *ERC20* token standard.

D. Severity grading of smart contract bugs

To give developers and researchers a clear understanding of the consequence of each bug, we grade the severity of each kind of bug. According to the *IEEE Standard Classification for Software Anomalies [23]*, we classify the effect of bugs into the following four categories:

- ***Functionality***. The required function cannot be performed correctly (or an unwanted function is performed).
- ***Performance***. Failure to meet performance requirements, such as rising operating costs.
- ***Security***. Failure to meet security requirements, such as failure of authority control, privacy breaches, property theft, etc.
- ***Serviceability***. Failure to meet maintainability requirements, such as reduced code readability.

According to the harmfulness of the above four effects, the grading criteria of these bugs are described as follows:

- **Critical**: These kinds of bugs must affect security.
- **High**: These kinds of bugs may affect security or necessarily affect functionality.
- **Middle**: These kinds of bugs may affect the functionality or necessarily affect performance.
- **Low**: These kinds of bugs may affect performance or necessarily affect serviceability.

According to the grading criteria, the severity level of each kind of bug is shown in Table II.

IV. *Jiuzhou*: A DATA SET FOR SMART CONTRACT BUGS

A. An overview of Jiuzhou data set

Jiuzhou provides examples of each kind of bugs to help smart contract researchers and developers better understand

TABLE II
A CLASSIFICATION OF SEVERITY LEVELS OF EACH KIND OF BUG

Name	Severity	Name	Severity	Name	Severity
A-a-ID	High	A-a-IO	High	A-a-IS	High
A-a-IT	High	A-a-W	High	A-b-HB	High
A-b-HS	High	A-b-I	High	A-c-UL	High
A-c-US	High	B-a-R	Middle	C-a-D	Middle
C-a-U	Middle	D-a-R	Critical	D-a-U	High
D-b-F	Middle	D-b-L	Critical	D-b-P	Middle
E-a-C	High	E-a-H	High	E-a-SA	High
E-a-SW	High	E-b-T	High	F-a-R	Middle
F-a-S	Middle	F-b -DBC	Middle	F-b -DBG	Middle
F-b -DBN	High	F-c-R	Middle	F-c-T	Middle
F-d-S	High	G-a-B	Low	G-a-II	Middle
G-a-IS	Middle	G-a-U	Middle	H-a-R	High
H-a-S	Critical	H-a-U	High	H-a-WC	Critical
H-a-WCN	High	H-b-N	High	H-b-P	High
I-a-I	Low	I-a-N	Low	I-a-T	High
I-a-UC	Low	I-a-UD	Low	I-b-I	Middle
I-b-F	Middle				

the bugs. The data set can be used as a benchmark to evaluate the abilities of smart contract analysis tools. *Jiuzhou* provides 176 smart contracts, covering all smart contract bugs counted in this paper. For each kind of bug, *Jiuzhou* provides at least a contract with the bug and a contract without the bug. Besides, we also provide some handwritten contracts used to induce analytical tools to make mistakes. The reason for providing these contracts is that in the existing research to evaluate the abilities of smart contract analysis tools, the test cases used for evaluation are all contracts that contain bugs. This will cause some tools to obtain precision rates that are inconsistent with their real performance (eg., static code scanning tool). To make the evaluation results more in line with the actual situation, we provide these contracts (we call these contracts *induced contracts*). We use the following strategies to construct induced contracts:

- **Premise**: Only context-related bugs have corresponding induced contracts, while statement-related bugs cannot have induced contracts (statement-related bug refers to the presence of the bug once a specific type of statement appears. eg., *nonstandard naming*, *byte[]*, *wrong operator*).
- **Strategy 1**: This strategy divides the statement (or structure) that causes the bug into multiple statements or multiple functions or multiple contracts. It can effectively reduce the precision of static code scanning tools.
- **Strategy 2**: This strategy provides the statements (or structures) that can fix the bug, but make the statements (or structures) invalid or unreachable. It can induce analysis tools to omit the bug.
- **Strategy 3**: This strategy uses uncommon means to fix the bug. It can induce analysis tools to misjudge.

B. Smart contract sources

We collect smart contracts from the following three sources:

- Other smart contract data sets (eg., [17], [18]). However, most of these contracts are developed using some older *Solidity* versions, we manually rewrite these smart contracts using the latest version of *Solidity* that contains these kinds of bugs.
- Sample code in the paper (eg. [5]), or sample code for smart contract audit checklists (eg. [37]). However, most of the sample code only contains one function or part of the contract, so we need to supplement these sample codes as a complete smart contract.
- We manually write smart contracts based on the features of these bugs. For some kinds of bugs with only text descriptions but no sample code, we manually write smart contracts that contain this kind of bug. According to the text description of the bug, we construct the behavior that causes the bug and supplement the code so that the bug exists in a complete contract that can be correctly compiled, and then verify that the bug in the contract will cause the consequences of the bug in the text description.

Table III shows the distribution of the number of unchanged smart contracts, modified smart contracts, and smart contracts that we developed manually.

TABLE III
NUMBER DISTRIBUTION OF THREE SMART CONTRACTS

	Unchanged smart contracts	Modified smart contracts	Handwritten smart contract
Num	21	69	86

C. Comparison with other data sets

All smart contracts of the *Jiuzhou* data set are developed using the latest version (0.4.26, 0.5.16, or 0.6.2) of *Solidity* containing bugs. Compared with several commonly used smart contract datasets [17], [18], [38], [39], *Jiuzhou* provides more smart contracts, uses the newer *Solidity* versions, and covers more kinds of smart contract bugs. A comparison of *Jiuzhou* with other commonly used data sets is shown in Table IV.

D. Possible use of the Jiuzhou data set

The *Jiuzhou* data set are useful for different kinds of developers:

- For smart contract developers, they can learn about smart contract bugs by reading these smart contracts.
- For smart contract analysis tool developers, the *Jiuzhou* data set can guide them to develop smart contract analysis tools and they can learn about smart contract programming patterns that are prone to false positives or omissions by reading induced contracts.

TABLE IV
COMPARISON OF *Jiuzhou* WITH OTHER DATA SETS

Data set	Number of contracts	Kinds of bugs	*Solidity version*
Jiuzhou	176	49	0.4.26 to 0.6.2
ethernaut [38]	21	21	0.4.18 to 0.4.24
not-so-smart -contracts [18]	25	12	0.4.9 to 0.4.23
SWC-registry [17]	114	33	0.4.0 to 0.5.0
capturetheether [39]	19	6	0.4.21

- For smart contract analysis tool evaluators, they can use these smart contracts as a benchmark to evaluate the abilities of smart contract analysis tools.

V. EVALUATION OF SMART CONTRACT ANALYSIS TOOLS

A. Overview

We use smart contracts in the *Jiuzhou* data set as a benchmark to evaluate the abilities of several smart contract analysis tools. An analysis tool with good ability should be able to analyze as many kinds of bugs as possible, and it also has good precision and recall rate. Therefore, we use the following indicators to measure the abilities of the analysis tools:

- **Coverage**. Coverage refers to the proportion of various bugs that can be detected by the analysis tool in the various bugs of *Jiuzhou* statistics. eg., *Oyente* claims to be able to detect 3 kinds of bugs (*A-a-IO*, *D-a-R*, *F-c-T*), and these three kinds of bugs are in *Jiuzhou* statistics, so *Oyente*'s coverage rate is 6% (3/49).
- **Precision and recall**. We use equation 1 and equation 2 to calculate the precision and recall rate. *tp* means that the tool analyzes the existence of the bug and the bug does exist. *fp* means that the tool analyzes the existence of the bug but the bug does not exist. *fn* means that the bug exists but the tool does not report the bug. The definitions of *tp*, *fp*, and *fn* are shown in Table V.

TABLE V
DEFINITION OF *tp*, *fp*, *fn*

Actual \ Analysis	**exist**	**non-exist**
exist	*tp*	*fn*
non-exist	*fp*	

$$Precision = (tp \div (tp + fp)) \tag{1}$$

$$Recall = (tp \div (tp + fn)) \tag{2}$$

We select the tools evaluated from the resources surveyed. Tools that meet the following two criteria will be evaluated by us: 1) The tool is free to use. 2) The tool takes a *Solidity* contract or compiled bytecode as input.

Table VI lists the 9 smart contract analysis tools we have selected. To our knowledge, the number of tools evaluated in this paper is not less than any existing work [15], [29], [40]. When installing these tools, we use the quick (or easy) method provided by the tools to install.

TABLE VI
THE SELECTED NINE SMART CONTRACT ANALYSIS TOOLS FOR EVALUATION

Tool	Maian [41], Mythril [42], Osiris [43], Oyente [44], Securify [45], Slither [46], SmartCheck [47], Remix[4] [48], SolidityCheck [49]

B. Coverage

We obtain the detected bugs according to the tools' documents. For the tools without detailed documents, we ask the developers via email. Fig 9 shows the coverage of various tools. The coverage of *Slither* is the highest, the coverage of static code scanning tools (eg., *SmartCheck*, *SolidityCheck*) is usually high, and the coverage of tools based on control flow (or data flow) analysis is usually low.

C. Precision and recall

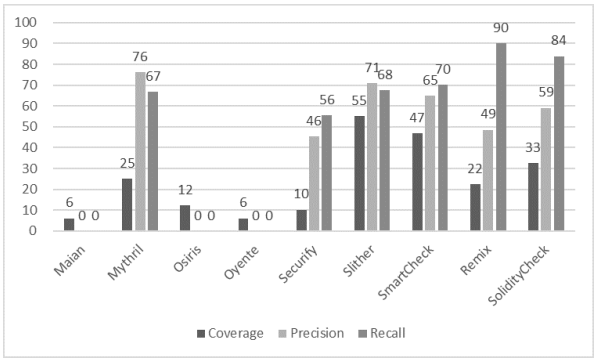

Fig. 9. *Coverage*, *recall* and *precision* of various tools

The test cases we chose are the smart contracts corresponding to the kinds of bugs that a tool claims to be able to detect. We use each analysis tool to test these contracts, and then calculate the recall and precision of each tool. The results are shown in Fig 9. Nine of the evaluated tools including six bytecode-based tools ([41], [42], [43], [44], [45], [46]), and only three of them can work normally, and the remaining three ([41], [43], [44]) cannot analyze the smart contracts of Solidity 0.4.26 and subsequent versions because they have not been updated for a long time, so no bugs are detected. This also reflects a real problem: when the Ethereum virtual machine (EVM) is updated, due to lack of update and maintenance motivation, the bytecode-based tools may not be able to adapt to the new version of the EVM, resulting in the limited availability of these tools. However, bytecode-based analysis tools usually use techniques such as control flow (data flow) analysis [50] and therefore have higher precision.

[4]We use the *solidity static analysis* of Remix

D. Result analysis

Fig 9 shows that *Slither* detects most kinds of bugs and has a good recall and precision rate. The analysis tool with the highest precision is *Mythril*, and *Remix* has the highest recall rate. Therefore, we recommend to use *Mythril*, *Slither*, and *Remix* for contract analysis, and the installations of these three tools are also very convenient (*Slither*: through *pip3* [51] installation, *Mythril*: through free plug-in in *Remix IDE* [31], and *Remix*: through *Remix IDE*).

Our evaluation results are similar to those of existing work of evaluating smart contract analysis tools [15], [29], [40]. *Mythril* and *Slither* are generally regarded as the two most effective tools. But our evaluation also has other interesting findings: *i)* Induced contracts are effective for the nine analysis tools (even tools based on symbolic execution). *ii)* The *solidity static analysis* function of *Remix* is worth evaluating because it may be the most commonly used and easy-to-use analysis tool for developers. According to our evaluation results, *Remix* should increase the kinds of bugs covered and try to improve the precision rate. *iii)* There are still 12 kinds of bugs that cannot be detected by any of these nine tools. *iv)* Although most analysis tools currently use techniques such as control flow (or data flow) analysis, static code scanning tools are still valuable, since the technical difficulty of developing static code scanning tools is usually low. Static code scanning tools usually cover more kinds of bugs, can adapt to the new version of *EVM* and *Solidity* faster, and has better coverage, recall, and availability.

VI. Related work

A. Statistics and classification of smart contract bugs

Some studies focus on the statistics and classification of smart contract bugs. Delmolino et al. [52] summarize four common smart contract programming pitfalls by investigating students' mistakes in learning smart contract programming. Atezi et al. [1] summarize 11 kinds of programming traps that may lead to security bugs. They believe that one of the main reasons for the continuous proliferation of smart contract bugs is the lack of inductive documentation for smart contract bugs. Chen et al. [14] collect smart contracts from *Stack Exchange* and Ethereum, define 20 kinds of code smell for smart contracts through manual analysis of smart contracts. Through interviews with smart contract developers, Zou et al. [53] reveal that smart contract developers still face many challenges when developing contracts, such as rudimentary development tools, limited programming languages, and difficulties in dealing with performance issue. Sayeed et al. [54] divide the attacks on Ethereum smart contracts into four categories according to the attack principle and introduce 7 kinds of smart contract bugs, and then they provide suggestions for implementing secure smart contracts. Dingman et al. [13] first count the existing bugs of Ethereum smart contract and then classify them using the *NIST* framework. They count 49 kinds of bugs and then classify 24 of them. Tikhomirov et al. [9] divide 20 kinds of smart contract bugs into security, functional, operational, and developmental, and give the severity of various bugs. *Smartdec* [16] divides the Ethereum smart contract bugs into three major categories: blockchain, language, and model. Their classification covers a total of 33 kinds of smart contract bugs. However, these studies have the following limitation: they only introduce some kinds of bugs and corresponding detect criteria.

B. Ethereum problem smart contract data sets

Some organizations and researchers provide problem smart contract data sets. *SmartContractSecurity* provides a list of smart contract bugs, including 33 kinds of bugs and problem smart contracts, but *SmartContractSecurity* does not classify these bugs, and some kinds of bugs also lack sample smart contracts [17]. *crytic* provides some examples of *Solidity* security issues covering 12 kinds of bugs, but 11 of them have not been updated for two years [18]. Durieux et al. [15] collect 47,587 Ethereum smart contracts, and then manually mark the smart contract bugs in 69 of these contracts, and then based on the smart contract bug classification provided by *DASP* [55], smart contract bugs in 69 contracts are divided into ten categories. But these smart contract data sets do not cover all kinds of smart contract bugs, and the number of smart contracts provided is relatively small.

C. Evaluate smart contract analysis tools

Some research focuses on evaluating the abilities of current smart contract analysis tools. Parizi et al. [29] evaluate four smart contract analysis tools using several commonly used data sets. Durieux et al. [15] develop an execution framework *SmartBugs* containing the smart contract analysis tools, and then they use *SmartBugs* to evaluate 9 smart contract analysis tools. Ghaleb et al. [40] implement *SolidFI*, a systematic method for automatically evaluating smart contract analysis tools. *SolidFI* first injects bugs into the contract, then runs smart contract analysis tools to detect bugs, and finally identifies false positives and omissions generated by the tool. Our evaluation results are similar to these work. Furthermore, as a complement to existing research we use the induced contracts as the test cases and also evaluate the performance of *Remix*. In this way, more interesting and useful conclusions are drawn.

VII. conclusion

By collecting the known bugs in smart contracts, we classify these bugs based on the extension of *IEEE Standard Classification for Software Anomalies* and give the detect criterion for each kind of bugs. Moreover, we construct a comprehensive smart contract dataset named *Jiuzhou* which covers all these bugs and provides the corresponding buggy smart contracts. Using the new dataset to evaluate the state-of-the-art tools for analyzing smart contracts, we obtained new observations.

References

[1] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International Conference on Principles of Security and Trust*. Springer, 2017, pp. 164–186.

[2] peckshield. (2020, May) Understanding *The DAO* accident. [Online]. Available: https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[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] 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.

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

[11] 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.

[12] J. Ye, M. Ma, T. Peng, Y. Peng, and Y. Xue, "Towards automated generation of bug benchmark for smart contracts," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2019, pp. 184–187.

[13] W. Dingman, A. Cohen, N. Ferrara, A. Lynch, P. Jasinski, P. E. Black, and L. Deng, "Classification of smart contract bugs using the nist bugs framework," in *2019 IEEE 17th International Conference on Software Engineering Research, Management and Applications (SERA)*, May 2019, pp. 116–123.

[14] 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.

[15] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," *2020 IEEE/ACM 42th International Conference on Software Engineering*, 2020.

[16] smartdec. (2020, Mar.) classification. [Online]. Available: https://github.com/smartdec/classification

[17] SmartContractSecurity. (2020, Jan.) Smart contract weakness classification and test cases. [Online]. Available: https://swcregistry.io/

[18] T. of Bits. (2020, Jan.) Examples of solidity security issues. [Online]. Available: https://github.com/crytic/not-so-smart-contracts

[19] Ethereum. (2020, Mar.) The ethereum wiki. [Online]. Available: https://github.com/ethereum/wiki

[20] ——. (2020, Mar.) Ethereum improvement proposals. [Online]. Available: https://eips.ethereum.org

[21] ——. (2020, Jan.) The development documents of solidity. [Online]. Available: https://solidity.readthedocs.io/en/v0.6.2/

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

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

[24] A. for Computing Machinery. (2020, Mar.) Acm digitai library. [Online]. Available: https://dl.acm.org/

[25] I. of Electrical and E. Engineers. (2020, Mar.) Ieee digital library. [Online]. Available: https://ieeexplore.ieee.org/Xplore/home.jsp

[26] Ethereum. (2020, Mar.) Ethereum github homepage. [Online]. Available: https://github.com/ethereum

[27] ——. (2020, Mar.) Ethereum foundation blog. [Online]. Available: https://blog.ethereum.org/

[28] ——. (2020, Mar.) Ethereum chatroom. [Online]. Available: https://gitter.im/orgs/ethereum/rooms/

[29] 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*. IBM Corp., 2018, pp. 103–113.

[30] G. Inc. (2019, Dec.) Open-source project repository. [Online]. Available: https://github.com/

[31] Ethereum. (2020, Mar.) Ethereum ide and tools for the web. [Online]. Available: http://remix.ethereum.org/

[32] V. Buterin. (2020, Mar.) Gas cost changes for io-heavy operations. [Online]. Available: https://eips.ethereum.org/EIPS/eip-150

[33] D. Siegel, "Understanding the dao attack," *Retrieved June*, vol. 13, p. 2018, 2016.

[34] Doingblock. (2020, Mar.) smart contract security. [Online]. Available: https://github.com/doingblock/smart-contract-security

[35] sec bit. (2020, Mar.) awesome-buggy-erc20-tokens. [Online]. Available: https://github.com/sec-bit/awesome-buggy-erc20-tokens/blob/master/ERC20_token_issue_list.md

[36] 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.

[37] knownsec. (2020, Jan.) Ethereum smart contracts security checklist from knownsec 404 team. [Online]. Available: https://github.com/knownsec/Ethereum-Smart-Contracts-Security-CheckList

[38] OpenZeppelin. (2020, Jan.) Web3/solidity based wargame. [Online]. Available: https://ethernaut.openzeppelin.com/

[39] SMARX. (2020, Mar.) Warmup. [Online]. Available: https://capturetheether.com/challenges/

[40] 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.

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

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

[43] christoftorres. (2020, Apr.) A tool to detect integer bugs in ethereum smart contracts. [Online]. Available: https://github.com/christoftorres/Osiris

[44] melonproject. (2020, Apr.) An analysis tool for smart contracts. [Online]. Available: https://github.com/melonproject/oyente

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

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

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

[48] Ethereum. (2020, Jan.) Browser-only ethereum ide and runtime environment. [Online]. Available: https://remix.ethereum.org/

[49] xf97. (2020, Apr.) Solioditycheck is a static code problem detection tool. [Online]. Available: https://github.com/xf97/SolidityCheck

[50] 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.

[51] T. P. community. (2020, May) The pypa recommended tool for installing python packages. [Online]. Available: https://pypi.org/project/pip/

[52] 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.

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

[54] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart contract: Attacks and protections," *IEEE Access*, vol. 8, pp. 24 416–24 427, 2020.

[55] N. Group. (2020, Mar.) Decentralized application security project. [Online]. Available: https://dasp.co/