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. Furthermore, there is neither a comprehensive classification framework for bugs of Ethereum smart contracts nor a smart contract data set supporting the framework, making it difficult for developers and researchers to fully understand the potential dangers in smart contracts. In this paper, to fill the gap, we first collect as many smart contract bugs as possible from multiple sources and then divide these bugs into 9 categories by extending the *IEEE Standard Classification for Software Anomalies*. Besides, we provide a set of smart contracts covering all kinds of bugs that we have counted. Developers can learn smart contract bugs from classification frameworks and data sets. Researchers can use our data set to evaluate existing smart contract analysis tools.**

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

I. Introduction

Blockchain [1] is a decentralized and distributed ledger whose data cannot be modified. It was first used as the underlying storage technology to support Bitcoin [1]. Then, Ethereum [2] introduced smart contracts to the blockchain, thereby expanding the scope of blockchain applications. Ethereum smart contracts are typically developed with high-level programming languages and then compiled into byte-code, which will be deployed to the blockchain through transactions.

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. For example, the *re-entrancy* vulnerability [3] in *the DAO* smart contract [4] led to a loss of $60 million.

The security of smart contracts has attracted the attention of many researchers, and several smart contract analysis tools have been developed to detect the potential bugs in smart contracts. However, recent research [5] shows that almost all smart contract analysis tools [6]–[14] can only detect some kinds of smart contract bugs. This cannot avoid the potential bugs in smart contracts, even smart contract analysis tools have been used. We believe that one of the reasons for this situation is the lack of a collection and classification of all the existing smart contract bugs, which makes developers lack guidance when smart contract analysis tools are developed. At the same time, recent studies showed that one main reason for the proliferation of smart contract bugs is the lack of a comprehensive classification framework for smart contract bugs [3]. Although a few studies summarized and classified some kinds of bugs in smart contracts [3], [15], [16], they have the following limitations:

- *The existing classification criteria for smart contract bugs is not comprehensive.* Dingman et al. [15] count 49 kinds of smart contract bugs and classify them using *NIST* framework, but only 24 kinds of bugs were classified accurately, and the remaining 25 kinds of bugs were classified into *other* category. Smartdec [17] divides smart contract bugs into three levels: blockchain, language and model, and classifies the bugs in each level, but their classification is not comprehensive enough, and some kinds of bugs (eg., *locked ether*) are not counted and classified.
- *Some kinds of bugs have been fixed officially.* Ethereum has already fixed some known bugs. For example, 13 kinds of bugs in [15] have been fixed and the *stack size limit* bug in [3] has also been fixed. Developers and researchers do not need to care about such bugs.
- *The existing data set for smart contract bugs is incomplete.* For example, *SmartContractSecurity* [18] counts 33 kinds of smart contract bugs, but only some kinds of bugs provide sample smart contracts; *crytic* [19] provides sample smart contracts for *Solidity* security issues, but only covers 12 kinds of bugs; Durieux et al. [20] provide a dataset containing 69 problematic smart contracts, but only covered 10 kinds of bugs. Moreover, these data sets use older language versions and provide fewer 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, networks, blogs, and related open-source projects, and finally obtain 323 records describing Ethereum smart contract bugs. Then, by reviewing the *Ethereum Wiki* [21], *Ethereum Improvement Proposals* [22] and the development documents of *Solidity* [23], we remove the bugs that had been fixed by Ethereum. We also merge the bugs caused by the same cause, 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. Finally, according to the classification framework, we provide a smart contract data set containing the collected bugs. 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 then classify them into 9 categories.
- We provide a data set of problematic smart contracts. The smart contracts in the data set to cover all the statistical bugs. It contains 176 smart contracts, including contracts that contain bugs, contracts that fix bugs and misleading contracts. By reading the smart contracts in the data set, smart contract 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 recommend a set of smart contract analysis tools. This set of smart contract analysis tools can detect as many kinds of bugs as possible, and has a good recall and accuracy.

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 and describes the characteristics of each category of bugs. Section 4 introduces the data set that matches the bug classification and gives the severity level of each bug. 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.

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 [2]. 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 [23]. *Solidity* 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 byte code of Ethereum virtual machine. *Solidity* provides many built-in symbols to perform various functions of Ethereum. For example, *transfer* and *send* are used to perform the transfer of ethers, and keywords such as *require* and *assert* are used to check the status. *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* language version used by the contracts.

C. IEEE Standard Classification for Software Anomalies

The *IEEE Standard Classification for Software Anomalies* [24] provides a unified method for the classification of traditional software anomalies. In its latest version, software anomalies are classified into six categories: *data*, *interface*, *logic*, *description*, *syntax*, *standards*. The standard also provides ranking criteria for the effect, severity, and priority of software anomalies. Researchers can flexibly tailor or extend this standard to adapt to different types of software. In this paper, we classify bugs in smart contracts based on the modification of this standard.

III. A CLASSIFICATION FRAMEWORK FOR SMART CONTRACT BUGS

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

A. Collect smart contract bugs

First, we collect smart contract bugs from academic literature, networks, 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* [25] and *IEEE Xplore digital library* [26]. The reason for the paper after 2014 was chosen is that Ethereum started ICO (initial coin offering) in 2014. For networks and blogs, we mainly focus on the *Github homepage of Ethereum* [27], *the development documents of Solidity* [23], *the official blogs of Ethereum* [28], *the Gitter chat room* [29], *Ethereum Improvement Proposals* [22] 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 [30]. Specifically, we use *smart contract bugs*, *smart contract problems*, *smart contract defects*, *smart contract vulnerabilities* and and *smart contract anomalies* as search keywords to retrieve related open-source projects on *GitHub* [31]. Besides, many smart contract analysis tool projects are also open-sourced on *GitHub* [31], and there are also some documents describing smart contract bugs in these projects. Therefore, we also use *smart contract analysis tools* and *smart contract security* as search keywords. We focus on the projects for Ethereum smart contracts. After removing duplicate search results, we obtained 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* feature of *Remix* [32]. Finally, from the resources mentioned above, we collected 323 records describing Ethereum smart contract bugs.

To continuously collect bugs, we expose various kinds of bugs on *Github* and accept other users to extend new bugs. Besides, we developed a crawler 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 standard, even if the names of the collected bugs are different, some similar 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 https://github.com/xf97/JiuZhou/blob/master/Correspondence.xlsx, which allows us to trace back the process of the merger.

C. Classify smart contract bugs

According to *IEEE Standard Classification for Software Anomalies [24]* issued in 2010, software anomalies are classified into six categories: *data*, *interface*, *logic*, *description*, *syntax*, *standards*. Among them, we do not consider *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*, *smart contract interactions*, *ethers exchange*, and *smart contract support software* are Ethereum-specific software anomalies. Consequently, the original classification provided by *IEEE Standard Classification for Software Anomalies [24]* cannot accurately classify these bugs. To accurately classify all kinds of bugs in smart contracts, we add four new categories: *security*, *performance*, *interaction*, and *environment*. 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 accounts.
5) ***Interface***. Bugs in specification or implementation of an interface.
6) ***Logic***. Bugs in decision logic, branching, sequencing, or 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 consulting *Ethereum Improvement Proposals* [22], *the Ethereum Wiki* [21] and the development documents of *Solidity* [23], we removed bugs that have been fixed by Ethereum (eg., the *call depth attack*, which was fixed in the *EIP150* [33]). Some of the merged 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. Any version of *Solidity* can result in bugs that do not have the *Solidity* versions listed. In the following, we list all categories of bugs lexicographically.

A. *Data*

The *data* category contains three sub-categories, and the definitions of the three sub-categories are shown in Table I.

TABLE I
DEFINITION OF THREE SUB-CATEGORIES OF *data*

Name	Definition
Calculation	Bugs due to integer calculations
Hidden	Bugs due to hidden variables or functions
Initialization	Bugs due to uninitialized variables

A-a. Calculation

1) *Integer division* (A-a-ID) [12]: All integer division results in *Solidity* are rounded down.

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

2) *Integer overflow and underflow* (A-a-IO) [6], [8], [18]: When the result exceeds the boundary value, the result will overflow or underflow.
3) *Integer sign* (A-a-IS) [8]: In *Solidity*, Converting *int* type to *uint* type (and vice versa) may produce incorrect results
4) *Integer truncation* (A-a-IT) [8]: Casting a long integer variable into a short integer variable may result in a loss of accuracy (eg. $uint256\ to\ uint8$).
5) *Wrong operator* (A-a-W) [18]: Before *Solidity* version 0.5.0, users can use =+ and =- operators in the integer operation without compiling errors (up to and including version 0.4.26).

A-b. Hidden

1) *Hidden built-in symbols* (A-b-HB) [34]: When a variable with the same name as a built-in symbol exists, the built-in symbol is hidden.
2) *Hidden state variables* (A-b-HS) [34]: Variables in a subclass will hide variables of the same name in the base class (up to and including version 0.5.16).
3) *Incorrect inheritance order* (A-b-I) [18]: *Solidity* supports multiple inheritances, and when the inheritance order is incorrect, the behavior of sub-classes may not be as expected by the developers (up to and including version 0.5.16).

A-c. Initialization

1) *Uninitialized local/state variables* (A-c-UL) [18]: 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).
2) *Uninitialized storage variables* (A-c-US) [18]: 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).

B. *Description*

The *description* category contains one sub-category, and the definitions of the one sub-category are shown in Table II.

TABLE II
DEFINITION OF ONE SUB-CATEGORY OF *description*

Name	Definition
Output	Bugs due to incorrect output information

B-a. Output

1) *Right-To-Left-Override control character* (U+202E) (B-a-R) [18]: Using *U+202E* characters will cause the output string to be inverted.

C. *Environment*

The *environment* category contains one sub-category, and the definitions of the one sub-category are shown in Table III.

C-a. Supporting software

TABLE III
DEFINITION OF ONE SUB-CATEGORY OF *environment*

Name	Definition
Supporting software	Bugs due to incorrect implementation of the *Solidity* compiler

1) *Delete dynamic array elements* (C-a-D) [35]: In *Solidity*, deleting dynamic array elements does not automatically shorten the length of the array and move the array elements.
2) *Using* continue-*statements in* do-while-*statements* (C-a-U) [23]: Before *Solidity* version 0.5.0, executing the *continue*-statements in the *do-while*-statements causes the condition decision statement to be skipped once (up to and including version 0.4.26).

D. *Interaction*

The *interaction* category contains two sub-categories, and the definitions of the two sub-categories are shown in Table IV.

TABLE IV
DEFINITION OF TWO SUB-CATEGORIES OF *interaction*

Name	Definition
Contract call	Bugs due to calls between contracts
Ether flow	Bugs due to contract receiving or sending ethers

D-a. Contract call

1) *Re-entrancy vulnerability* (D-a-R) [4], [14]: When the *call*-statement is used to call other contracts, the callee can call back the caller and enter the caller again. This mechanism creates the *re-entrancy vulnerability* when the following four characteristics exist in the contract:
   a) The *call-statement* is used to send ethers.
   b) The amount of *gas* to be carried is not specified.
   c) No callee's response function is specified.
   d) Ethers are transferred first and callee's balance is deduced later.
   - *Example*: We use an example to illustrate the *re-entrancy vulnerability*. In Fig 1, 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.

2) *Unhandled exception* (D-a-U) [6]: The contract can use low-level call statements such as *send*, *call*, and *delegatecall* to interact with other addresses. When a call made using these low-level call statements is abnormal, the call is not terminated and rollback, only *false* is returned.

D-b. Ether flow

1) *Forced to receive ether* (D-b-F) [18]: An attacker can force ethers to be sent to an address through self-destructing contracts or mining.
2) *Locked ether* (D-b-L) [9]: If the contract can receive ethers, but cannot send ethers, the ethers in the contract will be permanently locked.
3) *Pre-sent ether* (D-b-P) [17]: Malicious users can send ethers to the address of the 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.

E. *Interface*

The *interface* category contains two sub-categories, and the definitions of the two sub-categories are shown in Table V.

TABLE V
DEFINITION OF TWO SUB-CATEGORIES OF *interaction*

Name	Definition
Parameter	Bugs due to wrong parameters
Token Interface	Bugs due to wrong token contract interface

E-a. Parameter

1) *Call/delegatecall data/address is controlled externally* (E-a-C) [11], [36]: Contracts can call functions of other contracts. If the call data or address is controlled externally, the attacker can arbitrarily specify the call address, call function and parameters.
2) *Hash collisions with multiple variable length arguments* (E-a-H) [18]: Because *abi.encodePacked()* packs all parameters in order, regardless of whether the parameters are part of an array, the user can move elements within or between arrays. As long as all elements are in the same order, *abi.encodePacked()* will return the same result.
3) *Short address attack* (E-a-SA) [37]: 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.
   - *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 contract *A* code is shown in Fig 2.
     - *Step 2*: Jack buys 100 tokens of contract *A*, then registers for an Ethereum account with the last two digits zero (eg. 0x12345678901234567890123456789012345678900).
     - *Step 3*: Jack calls the function *SendCoin* with the given parameters, *_to*: 0x1234567890123456789012345678901234567 8 (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, in order 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.
4) *Signature with wrong parameter* (E-a-SW) [38]: In Fig 3, if the value passed is correct, the contract can authenticate with the *keccak256* and *ecrecover* functions. When the parameters of *ecrecover()* are incorrect, it will return the address 0x0. Assuming that the value of *_from* is also the 0x0 address, the check is bypassed, which means that anyone can transfer the tokens of the *0x0* address.

E-b. Token Interface

1) *Nonstandard token interface* (E-b-T) [39], [40]: Token contracts that do not meet *ERC20* [39], *ERC721* [40] and other token standards may have problems when interacting with other contracts.

F. *Logic*

The *logic* category contains four sub-categories, and the definitions of the four sub-categories are shown in Table VI.

TABLE VI
DEFINITION OF FOUR SUB-CATEGORIES OF *logic*

Name	Definition
Assembly code	Bugs due to improper use of assembly code
Denial of service	Bugs due to denial of service
Fairness	Bugs due to miners gaining a competitive advantage
Storage	Bugs due to overwrite storage

F-a. Assembly code

1) *Returning results using assembly code in the constructor* (F-a-R) [12]: Using assembly code return values in the constructor can make the contract deployment process inconsistent with developer expectations.

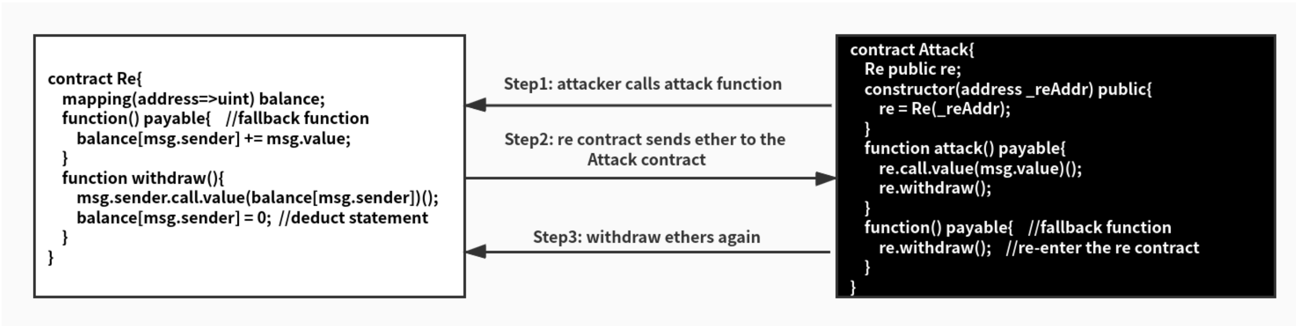

Fig. 1. An example of *re-entrancy vulnerability*

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

Fig. 2. Objective function of *short address attack*

```
//Calculate the signature of the public key _from
bytes32 hash = keccak256(_from,_spender,_value,nonce,name);
//Verify if it is the signature of _from
if(_from != ecrecover(hash,_v,_r,_s)) revert();
```

Fig. 3. Use *keccak256()* and *ecrecover()* to verify identity

2) *Specify function variable as any type* (F-a-S) [18]: Function variables can be specified as any type through assembly code (up to and including version 0.5.16).

F-b. Denial of service

1) *DOS by complex fallback function* (F-b-DBC) [11]: If the execution of the *fallback* function consumes more than 2300 *gas*, sending Ethers to the contract using *transfer* or *send* may fail.
2) *DOS by* gaslimit (F-b-DBG) [12], [35]: There is an attribute in the blocks of Ethereum, *gaslimit*, which specifies the upper limit of *gas* consumed by all transactions in the block. When a transaction consumes too much *gas*, the transaction may be refused to be packaged.
3) *DOS by non-existent address or malicious contract* (F-b-DBN) [12]: When the address that interacts with the contract does not exist, or the callee contract has an exception, the call will fail.

F-c. Fairness

1) *Results of contract execution affected by miners* (F-c-R) [6], [11]: The miner can control the attributes related to mining and blocks. If the functions of the contract depend on these attributes, the miner can interfere with the functions of the contract.
2) *Transaction order dependence* (F-c-T) [36]: Miners can decide which transactions are packaged into the blocks and the order in which transactions are packaged. 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.

F-d. Storage

1) *Storage overlap attack* (F-d-S) [36]: All data in the smart contract share common storage space. If the data is arbitrarily written into the storage, it may cause the data to overwrite each other.

G. *Performance*

The *performance* category contains one sub-category, and the definitions of the one sub-category are shown in Table VII.

TABLE VII
DEFINITION OF ONE SUB-CATEGORY OF *performance*

Name	Definition
Gas	Bugs due to unnecessary increase in *gas* consumption

G-a. Gas

1) *byte[]* (G-a-B) [23]: The *byte[]* type can act as a byte array, but due to padding rules, it wastes 31 *bytes* of space for each element. It is better to use the *bytes* type instead.
2) *Invariants in loop* (G-a-II) [12]: Placing the invariant in the loop causes extra *gas* consumption.
3) *Invariant state variables are not declared* constant (G-a-IS) [34]: The contract declares invariants but does not use the *constant* keyword to modify the invariants, which will cause more *gas* to be consumed.
4) *Unused public functions within a contracts should be declared external* (G-a-U) [34]: 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.

H. *Security*

The *security* category contains two sub-categories, and the definitions of the two sub-categories are shown in Table VIII.

H-a. Authority control

TABLE VIII
DEFINITION OF TWO SUB-CATEGORIES OF *security*

Name	Definition
Authority control	Bugs due to missing or incorrect authority controls
Privacy	Bugs due to privacy leaks

1) *Replay attack* (H-a-R) [18]: Since Ethereum's public chain has been forked many times, Ethereum now has many chains. Therefore, if the verification value used by a transaction can be predicted, an attacker can replay the transaction on another chain.
2) *Suicide contracts* (H-a-S) [9]: Authority control must be performed before a self-destructing operation, otherwise, the contract can be easily killed by an attacker.
3) Use *tx.origin* for authentication (H-a-U) [41]: *Solidity* provides the keyword *tx.origin* to indicate the initiator of the transaction. Do not use *tx.origin* for authentication. Because when an attacker deceives your trust, the attacker can trick you into sending a transaction to a malicious contract deployed by the attacker, and then the malicious contract forwards the transaction to your contract. At this point, the originator of the transaction is you, so the attacker can be authenticated.
4) *Wasteful contracts* (H-a-WC) [9]: A contract that anyone can withdraw the ethers is called a *wasteful contract*, and the reason for this bug is that the contract does not have authority control over the withdraw ethers.
5) *Wrong constructor name* (H-a-WCN) [12]: *Solidity* allows developers to use a function with the same name as the contract as a constructor. If the developers misspell the name of the constructor, it will make the constructor a public function that anyone can call (up to and including version 0.4.26).

H-b. Privacy

1) *Non-public variables are accessed by public/external functions* (H-b-N) [42]: *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.
2) *Public data* (H-b-P) [41]: For miners, all contract codes and the values of the state variables are visible, even if visibility is specified using *private* or *internal*.

I. Standard

The *standard* category contains two sub-categories, and the definitions of the two sub-categories are shown in Table IX.

I-a. Maintainability

1) *Implicit visibility level* (I-a-I) [43]: Failure to explicitly specify visibility can make the code difficult to understand (up to and including version 0.4.26. After version 0.4.26, functions must manually specify visibility, but state variables can still not specify visibility).

TABLE IX
DEFINITION OF TWO SUB-CATEGORIES OF *standard*

Name	Definition
Maintainability	Bugs due to reduced maintainability
Programming specification	Bugs due to violations of programming specifications

2) *Nonstandard naming* (I-a-N) [23]: *Solidity* specifies a standard naming scheme. Following the standard naming scheme will make the source code easier to understand.
3) *Too many digits* (I-a-T) [34]: Writing many consecutive numbers makes the code difcult to read and review. Developers can use scientific notation or exponential notation as an alternative.
4) *Unlimited compiler versions* (I-a-UC) [35]: In different versions of *Solidity*, different statements may have different semantics. When writing contracts, the *Solidity* version should be explicitly specified.
5) *Use deprecated built-in symbols* (I-a-UD) [12]: After *Solidity* version 0.5.0, several built-in symbols were discarded and replaced with other alternative built-in symbols (up to and including version 0.4.26).

I-b. Programming specification

1) view/constant *function changes contract state* (I-b-F) [12]: The keywords *view* and *constant* are provided in *Solidity* to modify functions, which means that these functions only read data from the blockchain without modifying the data. However, such rules are not mandatory, so developers can modify data in functions declared as *view* or *constant* (up to and including version 0.4.26).
2) *Improper use of* require*,* assert*, and* revert (I-b-I) [18]: *Solidity* provides several statements (*require*, *assert*, and *revert*) to handle errors. Although these statements are error handling statements, they are slightly different when used, so they need to be used correctly.

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 bug. According to the *IEEE Standard Classification for Software Anomalies [24]*, we classify the effect of software anomalies 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:

TABLE X

A CLASSIFICATION OF SEVERITY LEVELS OF EACH 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				

- **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 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 bug is shown in Table X.

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

A. *An overview of* Jiuzhou *data set*

Jiuzhou provides examples of each bug to help smart contract researchers and developers better understand the bugs and use these contracts as test cases to evaluate the capabilities of smart contract analysis tools. *Jiuzhou* provides 176 smart contracts, covering all smart contract bugs counted in this paper, including smart contracts containing bugs, smart contracts without bugs, and some misleading contracts that we manually write to mislead smart contract analysis tools.

For each kind of bug, *Jiuzhou* provides at least a contract with the bug and a contract without the bug. For certain contract-context related bugs, we provide *misleading contracts*. The role of *misleading contracts* is to induce smart contract analysis tools to misreport or omit. Currently, we only develop misleading contracts manually. In the future, we will study how to automatically generate misleading contracts.

B. *Smart contract sources*

We collect smart contracts from the following three sources:

- Other smart contract data sets (eg., [18], [19]).
- Sample code for smart contract analysis tool papers (eg. [8]), or sample code for smart contract audit checklists (eg. [43]). However, most of the sample code only contains the 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 characteristics of these bugs. For some kinds of bugs with only text descriptions but no sample code, we manually write smart contracts.

In addition to the sample codes obtained from smart contract analysis tool papers and smart contract audit checklists, we also modify some smart contracts collected from other smart contract data sets. Because these smart contracts of other data sets 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. Table XI shows the distribution of the number of unchanged smart contracts, modified smart contracts, and smart contracts that we developed manually.

TABLE XI

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 *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 [18], [19], [44], [45], *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 XII.

D. *Possible use of the* Jiuzhou *data set*

The *Jiuzhou* data set has the following possible uses:

TABLE XII

COMPARISON OF *Jiuzhou* WITH OTHER DATA SETS

Data set	**Number of contracts**	**Kinds of bugs**	***Solidity version***
Jiuzhou	176	49	0.4.24 to 0.6.2
ethernaut [44]	21	21	0.4.18 to 0.4.24
not-so-smart -contracts [19]	25	12	0.4.9 to 0.4.23
SWC-registry [18]	114	33	0.4.0 to 0.5.0
capturetheether [45]	19	6	0.4.21

- For smart contract developers,
  - **they can learn about smart contract bugs by reading these smart contracts.** Smart contract developers can read the contracts in the data set to understand these bugs, including programming patterns that may lead to bugs and ways to avoid them.
- For smart contract analysis tool developers,
  - **the *Jiuzhou* data set can guide them to develop smart contract analysis tools**. Current smart contract analysis tools can only detect some kinds of bugs [5]. This makes it possible that smart contracts still contain bugs even if smart contract developers have used smart contract analysis tools. *Jiuzhou* is the data set with the most kinds of bugs so far. According to our own experiences, smart contract analysis tools can be made to detect more kinds of bugs.
  - **they can learn about smart contract programming patterns that are prone to false positives by reading misleading contracts.** Smart contract analysis tool developers can improve the capabilities of smart contract analysis tools by reading misleading contracts to understand which programming patterns will cause analysis tools to make mistakes.
- For smart contract analysis tool evaluators, **they can use these smart contracts as a benchmark to evaluate the capabilities of smart contract analysis tools**. The smart contracts provided by *Jiuzhou* cover all kinds of possible bugs. Smart contract researchers can use these smart contracts as test cases to calculate the *recall*, *precision* and *analysis efficiency* of smart contract analysis tools.

V. Evaluation of smart contract analysis tools

A. Overview

We use smart contracts in the *Jiuzhou* data set as test cases to evaluate the capabilities of several smart contract analysis tools. An analysis tool with good capability should be able to analyze as many kinds of bugs as possible, and it has good accuracy and recall rate. We use the following indicators to measure the capabilities of analytical 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.
- **Accuracy and recall**. We use equation 1 and equation 2 to calculate the recall rate and accuracy. The definitions of *tp*, *fp*, and *fn* are shown in Table XIII.

TABLE XIII
Definition of *tp*, *fp*, *fn*

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

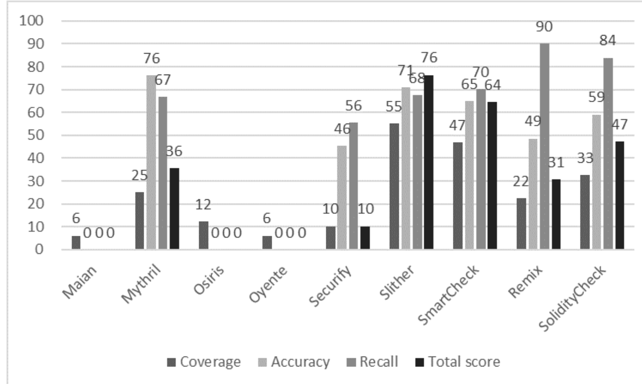

Fig. 4. Coverage, recall and accuracy of various tools

- **Total score**. The total score is calculated by equation 3. The product of the coverage and recall represents the actual recall rate, and the product of the coverage and accuracy represents the actual accuracy rate. The sum of the actual recall rate and the actual accuracy rate is used as an indicator of the capability of a tool.

$$Recall = (tp \div (tp + fn)) \tag{1}$$

$$Accuracy = (tp \div (tp + fp)) \tag{2}$$

$$Total\ score = Coverage \times (Recall + Accuracy) \tag{3}$$

We select smart contract analysis tools from the resources we surveyed. Table XIV lists the 9 smart contract analysis tools we have selected.

TABLE XIV
9 smart contract analysis tools as evaluation objects

Tool	Maian [46], Mythril [47], Osiris [48], Oyente [49], Securify [50], Slither [51], SmartCheck [52], Remix[2] [35], SolidityCheck [53]

B. Coverage

We obtain the detected bugs by consulting the relevant documents of the analysis tools. For the analysis tools of missing documents, we ask the developers via email. Fig 4 shows the coverage of various tools.

C. Accuracy and recall

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 accuracy of each tool. The results are shown in Fig 4.

The three tools *Maian*, *Osiris*, and *Oyente* cannot analyze the smart contract of Solidity 0.4.26 and subsequent versions, so no bug is detected, resulting in the recall and accuracy of these three tools of 0.

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

D. Total score and recommendation tools

According to equation 3, we calculate the total score of each tool, and the results are shown in Fig 4. Based on the total score, the best performing analysis tool is *Slither*. The analysis tool with the highest accuracy is *Mythril*, *Remix* with the highest recall. Based on the performance of each tool, we recommend using *Slither*, *SmartCheck*, *Mythril*, and *Remix* to analyze contracts together. This group of tools can cover 37 kinds of bugs. It is worth noting that there are still 12 kinds of bugs (eg., *integer truncation*) that cannot be detected by any of these 9 analysis tools, and developers have to manually check these kinds of bugs.

VI. Related work

A. Statistics of Ethereum smart contract bugs

Some studies have focused on statistical smart contract bugs. Destefanis et al. [54] propose the need to establish the blockchain software engineering by researching the accident of the freeze of the Ethereum parity wallet. Wohrer et al. [55] describe six kinds of smart contract security models that can be applied by smart contract developers to prevent possible attacks. Delmolino et al. [56] summarize four common smart contract programming pitfalls by investigating students' mistakes in learning smart contract programming. Atezi et al. [3] 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. [16] collect smart contracts from *Stack Exchange* and Ethereum, define 20 kinds of code smell for smart contracts through manual analysis of smart contracts. Wang et al. [57] propose a research framework for smart contracts based on a six-layer architecture and describe the bugs existing in smart contracts in terms of contract vulnerability, limitations of the blockchain, privacy, and law. Through interviews with smart contract developers, Zou et al. [58] 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 issues. Sayeed et al. [59] 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. Feist et al. [34] describe 45 kinds of smart contract bugs and implement *slither*, a static analysis tool of smart contract, to detect these bugs. However, these studies only provide statistics for smart contract bugs but do not classify smart contract bugs.

B. Category Ethereum Smart Contract Bugs

Some researches have contributed to the classification of smart contract bugs. Dingman et al. [15] 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. [12] divide 20 kinds of smart contract bugs into security, functional, operational, and developmental, and give the severity of various bugs. Zhang et al. [41] divide 20 kinds of smart contract bugs into three categories: security, functional, and potential threats in the code according to the hazards of the bugs, and develop a smart contract analysis tool *SolidityCheck* to detect these bugs. *Smartdec* [17] divides the Ethereum smart contract bugs into three major categories: blockchain, language, and model. Each major category contains several sub-classes, and each sub-class contains specific bugs. Their classification covers a total of 33 kinds of smart contract bugs. However, these classifications have three main limitations. First, they have not include all kinds of smart contract bugs, Second, they may include bugs that have been fixed officially. Third, they do not provide supporting smart contract data sets.

C. 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 [18]. *crytic* provides some examples of *Solidity* security issues covering 12 kinds of bugs, but 11 of them have not been updated for two years [19]. *OpenZeppelin* provides a wargame based on *Web3* and *Solidity* called *ethernaut* [44]. *ethernaut* contains 21 problem smart contracts, but *ethernaut* does not describe which bugs these smart contracts contain. Durieux et al. [20] 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* [60], smart contract bugs in 69 contracts are divided into ten categories. In general, these smart contract data sets do not cover all kinds of smart contract bugs, and the number of smart contracts provided is relatively small.

VII. conclusion

In this paper, we first count existing Ethereum smart contract bugs. Then we classify these bugs according to *IEEE Standard Classification for Software Anomalies*. Second, according to our bug statistics and classification, we provide a matching smart contract data set. Finally, we use the smart contracts in *Jiuzhou* data set as a benchmark to evaluate smart contract analysis tools, and recommend a set of smart contract analysis tools.

For future work, first we plan to generate misleading contracts automatically. Second, we try to study methods to automatically collect smart contract bugs and accurately classify them.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Manubot, Tech. Rep., 2019.

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

[3] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[16] J. Chen, X. Xia, D. Lo, J. Grundy, D. X. Luo, and T. Chen, "Domain specific code smells in smart contracts," *arXiv preprint arXiv:1905.01467*, 2019.

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

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

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

[20] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," *arXiv preprint arXiv:1910.10601*, 2019.

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

[22] ——. (2020, Mar.) Ethereum improvement proposals. [Online]. Available: https://eips.ethereum.org

[23] ——. (2020, Jan.) The development documents of solidity. [Online]. Available: https://solidity.readthedocs.io/en/v0.6.2/

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

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

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

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

[28] ——. (2020, Mar.) Ethereum foundation blog. [Online]. Available: https://blog.ethereum.org/

[29] ——. (2020, Mar.) Ethereum chatroom. [Online]. Available: https://gitter.im/orgs/ethereum/rooms/

[30] 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.

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

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

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

[34] 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.

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

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

[38] 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

[39] F. Vogelsteller. (2020, Jan.) Eip 20: Erc-20 token standard. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20/

[40] W. Entriken. (2020, Jan.) Eip 721: Erc-721 token standard. [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md

[41] P. Zhang, F. Xiao, and X. Luo, "Soliditycheck: Quickly detecting smart contract problems through regular expressions," *arXiv preprint arXiv:1911.09425*, 2019.

[42] quantstamp. (2020, Mar.) solidity-analyzer. [Online]. Available: https://github.com/quantstamp/solidity-analyzer

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

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

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

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

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

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

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

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

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

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

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

[54] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons, "Smart contracts vulnerabilities: a call for blockchain software engineering?" in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 19–25.

[55] M. Wohrer and U. Zdun, "Smart contracts: security patterns in the ethereum ecosystem and solidity," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 2–8.

[56] 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.

[57] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F.-Y. Wang, "Blockchain-enabled smart contracts: Architecture, applications, and future trends," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2019.
[58] 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.
[59] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart contract: Attacks and protections," *IEEE Access*, vol. 8, pp. 24 416–24 427, 2020.
[60] N. Group. (2020, Mar.) Decentralized application security project. [Online]. Available: https://dasp.co/