A Framework and Dataset for Bugs in Ethereum Smart Contracts

1st Pengcheng Zhang
College of Computer and Information, Hohai University
No.1 Xikang Road, NanjingJiangsu
Nanjing, P.R.China
pchzhang@hhu.edu.cn

2nd Feng Xiao
College of Computer and Information, Hohai University
No.1 Xikang Road, NanjingJiangsu
Nanjing, P.R.China
harleyxiao@foxmail.com

3rd Xiapu Luo
Department of Computing, the Hong Kong Polytechnic University
Mong Man Wai Building, Hong Kong SAR
HongKong, P.R.China
csxluo@comp.polyu.edu.hk

***Abstract*—Ethereum is the largest blockchain platform supporting smart contracts. Users usually deploy smart contracts by publishing the bytecode of smart contracts to the blockchain. Since the data in the blockchain cannot be modified, the deployed smart contract cannot be patched through code updates even if these contracts contain bugs. Recently, researchers proposed several smart contract analysis tools to detect bugs in smart contracts before deployment. Unfortunately, there is neither a comprehensive classification framework for Ethereum smart contract bugs nor a labeled dataset of smart contracts with bugs. In this paper, to fill in the gap, we first collect as many kinds of smart contract bugs as possible from several sources and then divided these bugs into 9 categories by extending the *IEEE Standard Classification for Software Anomalies*. Moreover, we provide a set of smart contracts containing the bugs in our framework. Users can use our data set to systematically evaluate existing smart contract analysis tools. Developers can learn smart contract bugs from the classification framework and data set to avoid bugs in smart contracts.**

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

I. Introduction

Blockchain [1] is a decentralized 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 in high-level programming languages and then compiled into byte-code, which will be deployed to the blockchain through a transaction.

Similar to other computer programs, it is difficult to avoid bugs in smart contracts. Recent years have witnessed various bugs in smart contracts, which resulted in huge losses. For example, the *re-entrancy* vulnerability [3] in *the DAO* smart contract [4] led to a lost of $60 million.

In recent years, smart contract analysis tools have been continuously developed. We noticed that almost smart contract analysis tools [5]–[13] can only detect a few smart contract bugs. No smart contract analysis tool can cover all existing smart contract bugs. This has led to the potential for bugs in smart contracts, even when using smart contract analysis tools. We believe that one of the reasons for this situation is the lack of a collection and classification of existing smart contract bugs, making developers lack guidance when developing smart contract analysis tools. 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], [14], [15], they have the following limitations:

- *Existing summarization of smart contract bugs are not comprehensive.* For example, Atezi et al. listed 11 kinds of bugs while Chen et al. [15] studied 20 kinds of code smells. Although Dingman et al. enumerated 49 kinds of bugs, only 24 of them were classified. Not all bugs were covered by them. For example, the *Transaction order dependence* bug is missing in [3], and the *Suicide contracts* bug is not included in [14], [15].
- *Some kinds of bugs have been fixed.* Ethereum has fixed some known bugs. For example, 13 kinds of bugs in [14] have been fixed and the *stack size limit* bug in [3] has also been fixed. Developers and users do not need to care about such bugs.
- *Missing a smart contract data set that complements the smart contract bug classification framework.* Such a data set can help smart contract developers and researchers better understand each bug and can be used as a benchmark for evaluating existing smart contract analysis tools.

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 and finally obtained 323 records describing Ethereum smart contract bugs. Then, by reviewing the *Ethereum Wiki*[1], *Ethereum Improvement Proposals*[2] and the development documents of *Solidity*[3], we remove the bugs that had been fixed by Ethereum. We also merge the bugs caused by the same cause, and 49 kinds of bugs left.

After that, by expanding the IEEE Standard Classification for Software Anomalies, we classified 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. Smart contracts in the data set are all written manually based on the characteristics of the bug or collected from the network.

The framework and dataset are called *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 by first collecting them from many sources and then classifying them into 9 categories.
- We provide a data set of problematic smart contracts. The smart contracts in the data set cover all the statistical bugs. It contains 176 smart contracts, including contracts that contain bugs, and contracts that fix bugs.

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 statistic, describes the characteristics of each bug, and gives the severity level of each bug. Section 5 describes the related work. Finally, Section 6 concludes the paper.

II. BACKGROUND

A. Smart contract

Smart contracts can be executed automatically on blockchain when certain conditions are met. In Ethereum, each smart contract and 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. *Solidity* is a Turing-complete high-level programming language capable of expressing arbitrarily complex logic. Smart contracts written in *Solidity* are compiled into the bytecode of the Ethereum virtual machine before deployment. *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. Therefore, 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* [16] provides a unified method for the classification of software anomalies. In the latest version of the standard, 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 software types.

As of now, the latest version of the standard is the 1044-2009.

III. A Classification Framework for Smart Contract Bugs

To build a comprehensive classification framework, we collecte 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, so we merge bugs based on the behavior of the bug. 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* and *smart contract problems* as search keywords to search for papers published since 2014 in *ACM digital library*[4] and *IEEE xplore digital library*[5]. The paper after 2014 was chosen because Ethereum started ICO (initial coin offering) in 2014. For networks and blogs, we mainly focus on the *Github* homepage of Ethereum [6], the development documents of *Solidity*[7], the official blogs of Ethereum[8], the Gitter chat room[9], *Ethereum Improvement Proposals*[10] and other resources.

Then, related open-source projects are also our focus, because the open-source community plays an important role in the field of software security [17]. Specifically, we use *smart contract bugs*, *smart contract problems*, *smart contract defects*, and *smart contract vulnerabilities* as search keywords

[1]https://github.com/ethereum/wiki
[2]https://eips.ethereum.org
[3]https://solidity.readthedocs.io/en/v0.6.2
[4]https://dl.acm.org/
[5]https://ieeexplore.ieee.org/Xplore/home.jsp
[6]https://github.com/ethereum/
[7]https://solidity.readthedocs.io/en/v0.6.2
[8]*https://blog.ethereum.org/*
[9]https://gitter.im/orgs/ethereum/rooms
[10]https://eips.ethereum.org

to retrieve related open-source projects on *GitHub* [18]. Besides, many smart contract analysis tool projects will also be open-sourced on *GitHub* [18], and there will also be 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.

Thirdly, the famous Ethereum smart contract analysis tool also detects 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*[11].

Finally, from the above resources, we collected 323 records describing Ethereum smart contract bugs.

In order to continuously collect bugs, we have opened various kinds of bugs on *Github*, and accept other users to extend new bugs. In addition, we developed a crawler called *BugGetter*[12]. *BugGetter* runs on a regular basis (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 bug collection 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, these bugs may point to the same bug. Consequently, we need to merge the duplicate bugs. The collected bugs generally have two attributes, namely, the behavior causing the bug and the consequence caused by the bug. If there is a bug *A*. Let,

- the behavior causing bug *A* be *b(A)*,
- the consequence 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 consequence of the bug. We merge *A* and *B*, rename the merged bug, summarize *c(A)* and *c(B)*, and give the consequence 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. After merging and deleting, 49 kinds of bugs are left.

We listed the correspondence of bugs before and after the merger in 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 [16]* issued in 2010, software anomalies are classified into six categories: *data*, *interface*, *logic*, *description*, *syntax*, *standards*. Among them, we ignore *syntax* category, because a smart contract with *syntax* bug cannot be compiled into bytecode and cannot be deployed in Ethereum, so it will not cause harm. Besides, the bugs caused by *gas*, *smart contract interactions*, *ethers exchange*, and *smart contract support software* are Ethereum-specific software anomalies, so the classification provided by *IEEE Standard Classification for Software Anomalies [16]* cannot accurately classify these bugs. In order to accurately classify all kinds of bugs, we add four new categories: *security*, *performance*, *interaction* and *environment*. Therefore, we divide smart contract bugs into the following nine categories:

1) ***Data***. Bugs in data definition, initialization, mapping, access, or use, as found in a model, specification, or implementation.
2) ***Description***. Bugs in description of software or its use, installation, or operation.
3) ***Environment***. Bugs due to errors in the supporting software.
4) ***Interaction***. Bugs that cause by contract 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 in 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.

In the process of merging bugs, by consulting *Ethereum Improvement Proposals*[13] and the development documents of *Solidity*[14], we removed bugs that have been fixed by Ethereum (eg., the *call depth attack*, which was fixed in the *EIP150* [19]). 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 will list the range of Solidity versions that caused these kinds of bugs.

[11]http://remix.ethereum.org/

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

[13]https://eips.ethereum.org

[14]https://solidity.readthedocs.io/en/v0.6.2

The following categories are listed in the lexicographical order.

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) [11].
   - *Cause*: All integer division results in *Solidity* are rounded down.
2) Integer overflow and underflow (A-a-IO) [5], [7], [20].
   - *Cause*: When the result exceeds the boundary value, the result will overflow or underflow.
3) Integer sign (A-a-IS) [7].
   - *Cause*: In *Solidity*, Converting *int* type to *uint* type (and vice versa) may produce incorrect results
4) Integer truncation (A-a-IT) [7].
   - *Cause*: 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) [20].
   - *Range of versions*: To Solidity version 0.4.26 (including this version).
   - *Cause*: Before *Solidity* version 0.5.0, users can use $=+$ and $=-$ operators in the integer operation without compiling errors.

A-b. Hidden

1) Hidden built-in symbols (A-b-HB) [21].
   - *Cause*: 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) [21].
   - *Range of versions*: To Solidity version 0.5.16 (including this version).
   - *Cause*: Variables in a subclass will hide variables of the same name in the base class.
3) Incorrect inheritance order (A-b-I) [20].
   - *Range of versions*: To Solidity version 0.5.16 (including this version).
   - *Cause*: *Solidity* supports multiple inheritances, and when the inheritance order is incorrect, the behavior of subclasses may not be as expected by the developers.

A-c. Initialization

1) Uninitialized local/state variables (A-c-UL) [20].
   - *Cause*: 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) [20]
   - *Range of versions*: To Solidity version 0.4.26 (including this version).
   - *Cause*: The uninitialized storage variable serves as a reference to the first state variable, which may cause the state variable to be inadvertently modified.

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) [20]
   - *Cause*: 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.

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

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

C-a. Supporting software

1) Delete dynamic array elements (C-a-D) [22]
   - *Cause*: In *Solidity*, deleting dynamic array elements does not automatically shorten the length of the array and move the array elements.
2) Using continue-statement in do-while-statement (C-a-U) [23]
   - *Range of versions*: To Solidity version 0.4.26 (including this version).
   - *Cause*: Before *Solidity* version 0.5.0, executing the a *continue-statement* in the *do-while-statements* causes the condition determination part to be skipped once.

D. ***Interaction***

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

D-a. Contract call

1) Re-entrancy vulnerability (D-a-R) [4], [13]

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

- *Cause*: When the *call-statement* is used to call other contracts, and the callee can call back the caller, thus entering 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 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 the *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. Then, the *deduct statement* of the contract *Re* will not be executed.

2) Unhandled exception (D-a-U) [5]
- *Cause*: The contract can use low-level call statements such as *send*, *call*, and *delegatecall* to interact with other addresses. When an exception occurs, the transaction is not terminated and rolled back, only *false* is returned.

D-b. Ether flow

1) Forced to receive Ether (D-b-F) [20]
- *Cause*: An attacker can force ethers to be sent to an address through self-destructing contracts or mining.

2) Locked Ether (D-b-L) [8]
- *Cause*: 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) [24]
- *Cause*: Malicious users can send ethers to the address of the contract before the contract is deployed.

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*

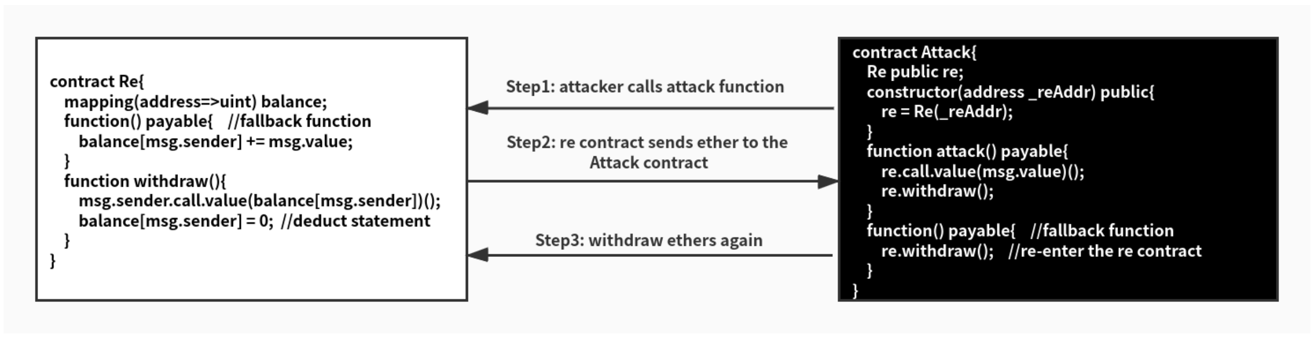
Name	Definition
Parameter	Bugs due to wrong parameters
Token Interface	Bugs due to wrong token contract interface

E-a. Parameter

1) Call/delegatecall data/address is controlled externally (E-a-C) [10], [25]
- *Cause*: Contracts can call functions of other contracts. If the call data or call 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) [20]
- *Cause*: 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.

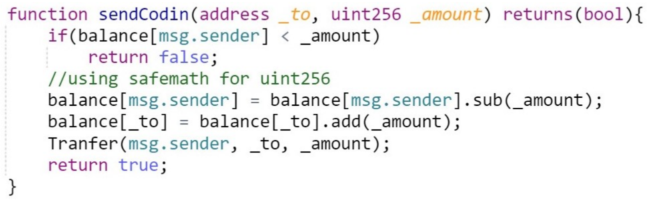
3) Short address attack (E-a-SA) [26]
- *Cause*: When Ethereum packs transaction data, if the data contains the address type and the length of the address type is less than 32 bits, the subsequent data is used to make up the 32 bits. We use an example to illustrate short address attacks:
- *Example*:
  – *Step1*: Tom deploys token contract *A* on Ethereum, which contains the *SendCoin* function. The function A code is shown in Fig 2.
  – *Step2*: Jack buys 100 tokens of contract *A*, and then registers for an Ethereum account with the last two digits zero (eg. 0x1234567890123456789012345678901234567800).
  – *Step3*: Jack calls the function *SendCoin* with the given parameters, *_to* : 0x12345678901234567890123456789012345678 (missing last two digits 0), *_amount* : 50.
  – *Step4*: 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


Step1: attacker calls attack function
Step2: re contract sends ether to the Attack contract
Step3: withdraw ethers again

```
contract Re{
   mapping(address=>uint) balance;
   function() payable{   //fallback function
      balance[msg.sender] += msg.value;
   }
   function withdraw(){
      msg.sender.call.value(balance[msg.sender])();
      balance[msg.sender] = 0;  //deduct statement
   }
}
```

```
contract Attack{
   Re public re;
   constructor(address _reAddr) public{
      re = Re(_reAddr);
   }
   function attack() payable{
      re.call.value(msg.value)();
      re.withdraw();
   }
   function() payable{   //fallback function
      re.withdraw();   //re-enter the re contract
   }
}
```

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

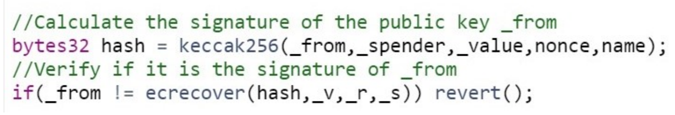


```
function sendCodin(address _to, uint256 _amount) returns(bool){
    if(balance[msg.sender] < _amount)
        return false;
    //using safemath for uint256
    balance[msg.sender] = balance[msg.sender].sub(_amount);
    balance[_to] = balance[_to].add(_amount);
    Tranfer(msg.sender, _to, _amount);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

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

transaction data is packed. Therefore, in order to make up for the bits 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) [**?**]
   - *Cause*: In Fig 3, if the value passed is correct, the contract can authenticate with these two 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 balance of the 0x0 address.



```
//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

E-b. Token Interface

1) Nonstandard token interface (E-b-T) [27], [28]
   - *Cause*: Token contracts that do not meet *ERC20 [27]*, *ERC721 [28]* 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) [11]
   - *Cause*: Using assembly code return values in the constructor can make the contract deployment process inconsistent with developer expectations.
2) Specify function variable as any type (F-a-S) [20]
   - *Range of versions*: To Solidity version 0.5.16 (including this version).
   - *Cause*: Function variables can be specified as any type through assembly code.

F-b. Denial of service

1) DOS by complex fallback function (F-b-DBC) [10]
   - *Cause*: If the execution of the *fallback* function consumes more than 2300 *gas*, it will result in failure to send ethers to the contract using *transfer* or *send*.
2) DOS by *gaslimit* (F-b-DBG) [11], [22]
   - *Cause*: There is an attribute in the blocks of Ethereum, *gaslimit*, which specifies the sum of the *gas* consumed by all transactions packed 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) [11]
   - *Cause*: 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) [5], [10]
   - *Cause*: The miner can control the attributes related to mining and blocks. If the features of the contract

depend on these attributes, the miner can interfere with the features of the contract.

2) Transaction order dependence (F-c-T) [25]
   - *Cause*: 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 result of this transaction, the miner may gain benefits by controlling the order in which the transactions are packaged.

F-d. Storage

1) Storage overlap attack (F-d-S) [25]
   - *Cause*: All data in the smart contract shares a 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]
   - *Cause*: The type *byte[]* is an array of bytes, 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) [11]
   - *Cause*: Placing invariants in loops causes extra *gas* consumption.
3) Invariant state variables are not declared constant (G-a-IS) [21]
   - *Cause*: The contract declares invariants but without using 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) [21]
   - *Cause*: 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

1) Replay attack (H-a-R) [20]
   - *Cause*: As the public chain of Ethereum has been forked many times, there are many chains in Ethereum now. Therefore, if the validation used by a transaction can be predicted, the attacker can replay the transaction on another chain.

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

2) Suicide contracts (H-a-S) [8]
   - *Cause*: Access control must be performed on the *self-destruct* operation, otherwise, the contract will be easily killed by the attacker.
3) Use *tx.origin* for authentication (H-a-U) [29]
   - *Level*: Ethereum protocol/virtual machine.
   - *Cause*: *Solidity* provides the keyword *tx.origin* to indicate the originator of the transaction. After the attacker deceives your trust, the attacker induces you to send a transaction to the attack contract deployed by the attacker, and the attack contract forwards the transaction to your contract. At this point, the originator of the transaction is you, so the attacker can authenticate.
4) Wasteful contracts (H-a-WC) [8]
   - *Cause*: A contract that says anyone can withdraw the ethers is called a *wasteful contract*, and the reason for this bug is that the contract does not have access control over the withdraw ethers.
5) Wrong constructor name (H-a-WCN) [11]
   - *Cause*: Before *Solidity* version 0.5.0, *Solidity* allows developers to use a function with the same name as the contract as the constructor. If the developers misspell the name of the constructor, it will make the constructor a public function that anyone can call.

H-b. Privacy

1) Non-public variables are accessed by public/external functions (H-b-NV) [30]
   - *Cause*: *Solidity* language requires visibility of specified state variables, of which *internal* and *private* specify that state variables can only be accessed internally. But *public/external* functions can still access *internal* and *private* state variables.
2) Public data (H-b-PD) [29]
   - *Cause*: Miners have complete network data backup. For miners, all contract codes and the values of the 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.

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

I-a. Maintainability

1) Implicit visibility level (I-a-I) [31]
   - *Range of versions*: To Solidity version 0.4.26 (including this version). After version 0.4.26, functions must manually specify visibility, but state variables can still not specify visibility.
   - *Cause*: Failure to explicitly specify visibility can make the code difficult to understand.
2) Nonstandard naming (I-a-N) [23]
   - *Cause*: *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) [21]
   - *Cause*: Writing many consecutive numbers makes the code difficult to read and review. Either scientific notation or exponential notation can be used.
4) Unlimited compiler versions (I-a-UC) [22]
   - *Cause*: In different versions of *Solidity*, different statements may have different semantics. When writing contracts, the language version should be explicitly specified.
5) Use deprecated built-in symbols (I-a-UD) [11]
   - *Range of versions*: To Solidity version 0.4.26 (including this version).
   - *Cause*: Aftet *Solidity* version 0.5.0, several built-in symbols were discarded and replaced with other alternative built-in symbols.

I-b. Programming specification

1) view/constant function changes contract state (I-b-F) [11]
   - *Range of versions*: To Solidity version 0.4.26 (including this version).
   - *Cause*: The keywords *view* and *constant* (before *Solidity* version 0.5.0) 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*.
2) Improper use of *require*, *assert*, and *revert* (I-b-I) [20]
   - *Cause*: *Solidity* language provides several statements (*require*, *assert*, and *revert*) to handle errors. These statements 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 [16]*, we classify the effect of software anomalies into the following four categories:

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

According to the harmfulness of the above four impacts, the grading criteria are described as follows:

- **Critical**: These kinds of bugs must affect security.
- **Major**: These kinds of bugs may affect security or necessarily affect functionality.
- **Minor**: These kinds of bugs may affect functionality or necessarily affect performance.
- **Inconsequential**: These kinds of bugs may affect performance or necessarily affect maintainability.

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 misleading contracts that we manually write to mislead smart contract analysis tools. The data set is available at: https://github.com/xf97/JiuZhou. *Jiuzhou* also uses a json [32] file to record the number of bug lines in the problem smart contract. Researchers can use the program to parse the json file to easily obtain the location of each bug.

B. Smart contract sources

We collect smart contracts from the following three sources:

- Other smart contract data sets (eg., [20], [33]).
- Sample code for smart contract analysis tool papers (eg. [7]), or sample code for smart contract audit checklists (eg. [31]). However, most examples only contain a function or part of the contract, so we need to rewrite these examples into the complete smart contracts.
- We manually write smart contracts based on the characteristics of the bug. For some kinds of bugs with only text descriptions but no sample code, we manually write smart contracts based on the characteristics of the bugs.

TABLE X
A CLASSIFICATION OF SEVERITY LEVELS OF EACH BUG

Name of bug	**Severity**	**Name of bug**	**Severity**
A-a-ID	Major	A-a-IO	Major
A-a-IS	Major	A-a-IT	Major
A-a-W	Major	A-b-HB	Major
A-b-HS	Major	A-b-I	Major
A-c-UL	Major	A-c-US	Major
B-a-R	Minor	C-a-D	Minor
C-a-U	Minor	D-a-R	Critical
D-a-U	Major	D-b-F	Minor
D-b-L	Critical	D-b-P	Minor
E-a-C	Major	E-a-H	Major
E-a-SA	Major	E-a-SW	Major
E-b-T	Major	F-a-R	Minor
F-a-S	Minor	F-b-DBC	Minor
F-b-DBG	Minor	F-b-DBN	Major
F-c-R	Minor	F-c-T	Minor
F-d-S	Major	G-a-B	Inconsequential
G-a-II	Minor	G-a-IS	Minor
G-a-U	Minor	H-a-R	Major
H-a-S	Critical	H-a-U	Major
H-a-WC	Critical	H-a-WCN	Major
H-b-NV	Major	H-b-PD	Major
I-a-I	Inconsequential	I-a-N	Inconsequential
I-a-T	Major	I-a-UC	Inconsequential
I-a-UD	Inconsequential	I-b-I	Minor
I-b-F	Minor		

In addition to the sample code obtained from smart contract analysis tool papers and smart contract audit checklists, we also rewrite some smart contracts collected from other smart contract data sets. Because these smart contracts of other data sets were 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, rewritten smart contracts, and smart contracts that we developed manually.

TABLE XI
NUMBER DISTRIBUTION OF THREE SMART CONTRACTS

	Unchanged smart contracts	**Rewritten smart contracts**	**Handwritten smart contract**
Num	21	69	86

C. *Structure of the data set*

The structure of the *Jiuzhou* data set is shown in Fig 4.

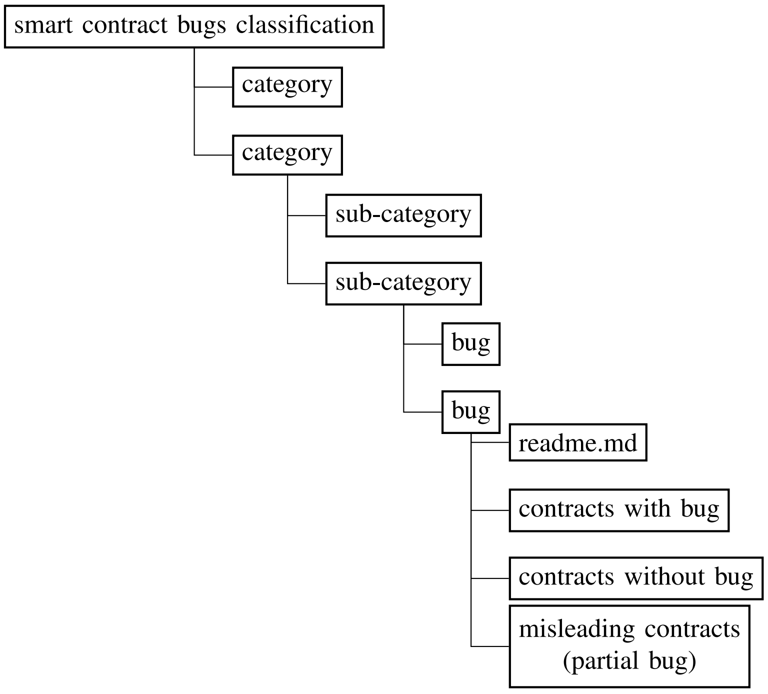


Fig. 4. Structure of the *Jiuzhou* data set

The structure of the *Jiuzhou* data set is consistent with the classification framework. 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*. Based on retaining the statements that may introduce these kinds of bugs, we change the contract context to avoid these kind of bugs. These contracts are provided to introduce errors in smart contract analysis tools.

Unfortunately, we can now only develop misleading contracts manually. In the future we will study how to automatically generate misleading contracts.

D. *Comparison with other datasets*

All smart contracts of *Jiuzhou* data set are developed using the latest version (0.4.26, 0.5.16 or 0.6.2) of the *Solidity* containing bugs. Besides the smart contract source-code files, *Jiuzhou* also provides compiled bytecode files.

Compared with several commonly used smart contract datasets [20], [33]–[35], *Jiuzhou* provides more smart contracts, uses the newer language versions, and covers more kinds of smart contract bugs. A comparison of *Jiuzhou* with other commonly used data sets is shown in Table XII.

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

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

- Learn about smart contract bugs by reading these smart contracts.

[15]https://ethernaut.openzeppelin.com
[16]https://github.com/crytic/not-so-smart-contracts
[17]https://github.com/SmartContractSecurity/SWC-registry
[18]https://capturetheether.com/challenges/

TABLE XII
COMPARISON OF *Jiuzhou* WITH OTHER DATA SETS

Data set	**Number of contracts**	**Kinds of bugs covered**	***Solidity version***
Jiuzhou	176	49	0.4.24 to 0.6.2
ethernaut[15]	21	21	0.4.18 to 0.4.24
not-so-smart-contracts[16]	25	12	0.4.9 to 0.4.23
SWC-registry[17]	114	33	0.4.0 to 0.5.0
capturetheether[18]	19	6	0.4.21

- Use these smart contracts as test cases to evaluate the capabilities of smart contract analysis tools.

V. RELATED WORK

A. Ethereum problem smart contract dataset

Some organizations and research institutions 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 supporting smart contracts [20]. *crytic* provides some examples of *Solidity* security issues covering 12 bugs, but 11 of them were last updated two years ago [33]. *OpenZeppelin* provides a wargame based on *Web3* and *Solidity* called *ethernaut* [34]. *ethernaut* contains 21 problem smart contracts, but *ethernaut* does not describe which bugs these smart contracts contain.

B. Statistics and investigation of smart contract bugs

The endless stream of Ethereum smart contract accidents has attracted researchers' attention, and some studies have focused on statistical smart contract bugs. Destefanis et al. [36] propose the need to establish the blockchain software engineering by researching the accident of the freeze of the Ethereum parity wallet. Wohrer et al. [37] describe six kinds of smart contract security models that can be applied by smart contract developers to prevent possible attacks. Delmolino et al. [38] summarize four common smart contract programming pitfalls by investigating the mistakes students when they are teaching 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. Dingman et al. [14] first count the existing bugs of Ethereum smart contract, and then classify them using *NIST* framework. They counted 49 kinds of bugs, and then classified 24 of them. Chen et al. [15] collect smart contracts from *Stack Exchange* and Ethereum, define 20 kinds of code smells for smart contracts through manual analysis of smart contracts. Wang et al. [39] 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. [40] 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.

In general, there are two main limitations in these work. First, there are no enough comprehensive statistics of existing smart contract bugs. Second, a data set supporting the statistical results is still missing.

C. Smart contract analysis tool

Some researchers focus on developing automation tools to detect smart contract bugs, and from their work we have learned many smart contract bugs. Luu et al. [5] develop *Oyente* by using symbol execution. *Oyente* can detect four kinds of security bugs: *unhandled exception*, *transaction order dependence*, *timestamp dependence* and *re-entrancy vulnerability*. Kalra et al. [6] implement *ZEUS*. *ZEUS* can detect seven kinds of smart contract bugs, and four of them are the same as *Oyente* and other three kinds of bugs are: *unchecked send*, *failed send*, and *integer overflow/underflow*. Jiang et al. [10] use fuzzy testing to detect smart contract bugs, and implement a tool called *ContractFuzzer*. *Contractfuzzer* can detect seven kinds of smart contract bugs, which are: *gasless send*, *unhandled exception*, *re-entrancy vulnerability*, *timestamp dependency*, *block number dependency*, *dangerous delegatecall* and *freezing ether*. Zhang et al. [29] count 20 kinds of smart contract bugs and divided them into three categories: *security*, *performance*, and *potential threats*. Then, they use regular expressions and program instrumentation to implement *Soliditycheck*. Experiments show that *Soliditycheck* has very high analysis efficiency. Chen et al. [9] investigate and find that the recommended smart contract compilers may generate bytecode containing expensive patterns. Consequently, they implement *Gasper*, a symbol-based execution tool for detecting expensive patterns in bytecode.

However, due to the lack of a comprehensive statistics of smart contract bugs, these smart contract analysis tools can only detect part smart contract bugs. This makes it possible that contracts still contain other bugs even if smart contracts are detected using these analysis tools. Different to these work, our goal is to provide a comprehensive classification framework and data set for smart contract bugs.

VI. CONCLUSION

In this paper, we first count existing Ethereum smart contract bugs, and merge duplicate bugs based on the behaviors that caused these bugs. Then we classify these bugs according to *IEEE Standard Classification for Software Anomalies*. Finally, according to our bug statistics, we provide a matching smart contract data set. As far as we know, it is the most comprehensive smart contract bug statistics till now.

For future work, first, we will try to generate misleading contracts automatically. Second, we will 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] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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

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

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

[21] 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.

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

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

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

[25] 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.

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

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

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

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

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

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

[32] A. A. Frozza, R. d. S. Mello, and F. d. S. d. Costa, “An approach for schema extraction of json and extended json document collections,” in *2018 IEEE International Conference on Information Reuse and Integration (IRI)*, 2018, pp. 356–363.

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

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

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.