A Framework and Dataset for Bugs in Ethereum Smart Contracts

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 they have bugs. Recently, researchers proposed several smart contract analysis tools to detect bugs in smart contracts before deployment. Unfortunately, there is neither a comprehensive framework for bugs in smart contracts nor a labelled 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 propose a framework for them. Moreover, we create a set of smart contract containing the bugs in our framework. Users can leverage our dataset for systematically evaluating existing smart contract analysis tools and developers can learn from the framework and the problematic smart contracts to avoid the bugs in their smart contracts.

CCS CONCEPTS

• Security and privacy; **• Software and its engineering → Software testing and debugging**;

KEYWORDS

Blockchain, Ethereum, Solidity, Smart contract bug

ACM Reference Format:

. 2018. A Framework and Dataset for Bugs in Ethereum Smart Contracts. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Blockchain [21] is a decentralized distributed ledger whose data cannot be modified. It was first used as the underlying storage technology to support Bitcoin [21]. Then, Ethereum [2] enables executing Turing-complete programs in terms of smart contracts on blockchain. Smart contracts are typically developed in high-level programming languages and then compiled into bytecode, 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 [1] in *the DAO* smart contract [28] led to a lost of $60 million.

In recent years, smart contract analysis tools have been continuously developed. We noticed that most smart contract analysis tools [4, 13, 16, 17, 19, 20, 22, 32, 33] 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 [1]. Although a few studies summarized and classified some kinds of bugs in smart contracts [1, 3, 7], 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. [3] 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 [1], and the *Suicide contracts* bug is not included in [3, 7].
- *Some kinds of bugs have been fixed.* Ethereum has fixed some known bugs. For example, 13 kinds of bugs in [7] have been fixed and the *stack size limit* bug in [1] has also been fixed. Developers and users do not need to care about such bugs.
- *There lacks of a dataset of problematic smart contracts with those bugs.* Such dataset can help smart contract developers and researchers better understand each bug, and serve as a benchmark for evaluating existing smart contract analysis tools.

In this paper, to fill in the gap, we first carefully collect known bugs in Ethereum smart contracts from many sources, including, academic literature, networks, blogs, and related open-source projects, and eventually obtain 117 kinds of bugs. Then, by reviewing the *Ethereum Wiki*[1] and the development documents of *Solidity*[2], we remove the bugs that had been fixed by Ethereum. We also merge the bugs caused by the same behavior, and 50 kinds of bugs left. After that, we propose a framework by classifying the 50 kinds of bugs into 20 categories according to the reasons that cause the bugs. Finally, according to the new framework, we construct a dataset of smart contracts containing the collected bugs. More precisely, we develop these problematic smart contracts according to the characteristics of each bug. 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 20 categories.

[1]https://github.com/ethereum/wiki
[2]https://solidity.readthedocs.io/en/v0.6.2

- We construct a dataset of problematic smart contracts with the identified bugs. It contains 174 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 bug statistics and classification framework, describes the characteristics of each bug, and gives the severity level of each bug. Section 4 introduces the dataset that matches the bug statistics. Section 5 describes the related work. Finally, Section 6 concludes the paper.

2 BACKGROUND

2.1 Smart contract

Smart contracts can be executed automatically on blockchain when certain conditions are met. In Ethereum, each smart contract and user are 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.

2.2 Solidity

Solidity is the most widely used programming language for developing 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.

2.3 IEEE Standard Classification for Software Anomalies

The earlier an anomaly is found in the software life cycle, the lower the cost of fixing the anomaly, and often the easier it is. This encourages the use of tools, techniques and methods to find anomalies faster. To assess the operation of these tools, techniques and methods, standard anomaly classification data is required. To this end, *IEEE*[3] develops *IEEE Standard Classification for Software Anomalies* that provide a unified method for the classification of software anomalies, regardless of when they originated, or encountered during the life cycle of a project, product, or system. Classification data can be used for a variety of purposes, including defect cause-effect analysis, project management, and software process improvement.

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

[3]https://www.ieee.org/

3 A FRAMEWORK FOR SMART CONTRACT BUGS

3.1 Overview

To make a comprehensive framework, we collect smart contract bugs from many sources, including academic literature, networks, blogs, and related open-source projects. Since there is no uniform bug naming standard, same bugs may have different names, and thus we merge the bugs according to their behaviors. We further divide the merged bugs into 20 categories according to the causes of the bugs, and give each bug a severity level according to its consequence.

3.2 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 *Google scholar*. 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 [4], the development documents of *Solidity*[5], the official blogs of Ethereum[6], the Gitter chat room[7], *Ethereum Improvement Proposals*[8] 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 [25]. Specifically, we use *smart contract bugs*, *smart contract problems*, *smart contract defects*, and *smart contract vulnerabilities* as search keywords to retrieve related open-source projects on *GitHub* [15].In addition, many smart contract analysis tool projects will also be open sourced on *GitHub* [15], 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 well-known Ethereum smart contract analysis tools will detect bugs in smart contracts. 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*[9].

Finally, from the above resources, we collected 117(to do) smart contract bugs with different names.

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*[10]. *BugGetter* runs on a regular basis (now set to 15 days, adjustable), and sends a query request 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*,

[4]https://github.com/ethereum/
[5]https://solidity.readthedocs.io/en/v0.6.0
[6]*https://blog.ethereum.org/*
[7]https://gitter.im/orgs/ethereum/rooms
[8]https://eips.ethereum.org
[9]http://remix.ethereum.org/
[10]https://github.com/xf97/BugGetter

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.

3.3 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, 50 kinds of bugs are left.

We list the bugs before and after the merger and their corresponding relationships in Appendix A, so that we can trace back to the source of the bug after the merger.

3.4 Classify smart contract bugs

According to *IEEE Standard Classification for Software Anomalies*[14] issued in 2010, software anomalies are classified into the following six categories:

(1) ***Data***. Bugs in data definition, initialization, mapping, access, or use, as found in a model, specification, or implementation.
(2) ***Interface***. Bugs in specification or implementation of an interface.
(3) ***Logic***. Bugs in decision logic, branching, sequencing, or computational algorithm, as found in natural language specifications or in implementation language.
(4) ***Description***. Bugs in description of software or its use, installation, or operation.
(5) ***Syntax***. Nonconformity with the defined rules of a language.
(6) ***Standards***. Nonconformity with a defined standard.

We have classified 50 bugs according to the *IEEE Standard Classification for Software Anomalies*. Since the emergence of Ethereum smart contracts is later than this version of the *IEEE Standard Classification for Software Anomalies*, the bugs caused by *gas* and *smart contract interaction* cannot be classified well into the existing bug categories, so we have added the following four bug categories:

(1) ***Security***. Bugs that threaten contract security, such as authentication, privacy/confidentiality, property.
(2) ***Performance***. Bugs that cause increased *gas* consumption.
(3) ***Interaction***. Bugs caused by contract interaction with other accounts.
(4) ***Environment***. Bugs due to mistakes in the software that supports Ethereum smart contracts.

Ethereum smart contract contract bugs come from four levels (blockchain, Ethereum protocol/virtual machine, smart contract programming language, developer mistakes), so we also divide bugs into four levels. The purpose of layering is to indicate the duration of each bug:

- *Blockchain*. This level of bugs is caused by blockchain technology. If Ethereum does not optimize the underlying blockchain technology, this level of bugs will continue to exist.
- *Ethereum protocol/virtual machine*. This level of bugs is caused by Ethereum protocol/virtual machine. This level of bugs will be fixed with the update (*hard fork*) of Ethereum. We obtained each Ethereum update from the *Ethereum Improvement Proposals*[11], and deleted all the bugs that had been fixed, so this level of bugs were not fixed.
- *Smart contract programming language*. This level of bugs is caused by smart contract programming language (*Solidity*). Because developers can use multiple versions of the *Solidity* language to write smart contracts, we will specify the range of versions of each bug.
- *Developer mistakes*. This level of bugs is caused by Developer mistakes and needs to be fixed by the developer by carefully reviewing the code.

The following categories are listed in the lexicographical order.

A. ***Data***

A-a. Calculation

(1) Integer division (A-a-ID) [32].
    - *Level*: Smart contract programming language.
    - *Range of versions*: Not fixed.
    - *Cause*: Until now, *Solidity* does not support decimals and fixed-point numbers. Consequently, all integer division results are rounded down.
(2) Integer overflow (A-a-IO) [20, 30, 33].
    - *Level*: Smart contract programming language.
    - *Range of versions*: Not fixed.
    - *Cause*: There are also integer overflow and underflow in Ethereum. When integer overflow or underflow occurs, Ethereum will not throw an exception.
(3) Integer sign (A-a-IS) [33].
    - *Level*: Smart contract programming language.
    - *Range of versions*: Not fixed.
    - *Cause*: In *Solidity*, converting a negative integer to an unsigned integer will result in an incorrect conversion result and will not throw an exception.
(4) Integer truncation (A-a-IT) [33].
    - *Level*: Smart contract programming language.
    - *Range of versions*: Not fixed.

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

- *Cause*: Because the representation range of a short integer is smaller than that of a long integer, casting a long integer variable into a short integer variable may result in a loss of accuracy (eg. *uint*256 *to uint*8).

(5) Wrong operator (A-a-W) [30].
- *Level*: Smart contract programming language.
- *Range of versions*: To Solidity version 0.4.26 (including this version).
- *Cause*: The wrong operators still compile correctly. In our practice, we use compilers before *Solidity* version 0.5.0 and then use = + and = − operators in the contract without compiling errors.

A-b. Hidden

(1) Hidden built-in symbols (A-b-HB) [12].
- *Level*: Smart contract programming language.
- *Range of versions*: Not fixed.
- *Cause*: The wrong operators still compile correctly. In our practice, we use compilers before *Solidity* version 0.5.0 and then use = + and = − operators in the contract without compiling errors.

(2) Hide state variables (A-b-HS) [12].
- *Level*: Smart contract programming language.
- *Range of versions*: To Solidity version 0.5.16 (including this version).
- *Cause*: Inheritance makes a common bug that causes state variables in sub-classes to hide state variables with the same name in the base class.

(3) Incorrect inheritance order (A-b-I) [30].
- *Level*: Smart contract programming language.
- *Range of versions*: To Solidity version 0.5.16 (including this version).
- *Cause*: *Solidity* supports multiple inheritances, and when the subclass has functions or variables of the same name in the base classes, the order of inheritance matters determines which one will be integrated into the subclass.

A-c. Initialization

(1) Uninitialized local/state variables (A-c-UL) [30].
- *Level*: Smart contract programming language.
- *Range of versions*: Not fixed.
- *Cause*: If local/state variables are declared but not initialized, these state variables are set to the default values. For example, an uninitialized *address* type will be assigned the default value (*address(0x0)*), and sending ethers to this address will cause ethers to be destroyed.

(2) Uninitialized storage variables (A-c-US) [30]
- *Level*: Smart contract programming language.
- *Range of versions*: To Solidity version 0.4.26 (including this version).
- *Cause*: Of all the uninitialized bugs, the uninitialized storage variable is the most dangerous because the uninitialized storage variable serves as a reference to the first state variable.

B. ***Description***

B-a. Output

(1) Right-To-Left-Override control character (U+202E) (B-a-R) [30]
- *Level*: Smart contract programming language.
- *Range of versions*: Not fixed.
- *Cause*: When printing *U+202E* characters [11], the character string will be inverted. In some cases, this can cause the true intention of the contract to be hidden.

C. ***Environment***

C-a. Supporting software

(1) Delete dynamic array elements (C-a-D) [9]
- *Level*: Smart contract programming language.
- *Range of versions*: Not fixed.
- *Cause*: Before *Solidity* version 0.5.0, using the *do-while-statements* can cause a bug. Executing the a *continue-statement* in the *do-while-statements* causes the condition determination part to be skipped once.

(2) Using continue-statement in do-while-statements (C-a-U) [29]
- *Level*: Smart contract programming language.
- *Range of versions*: To Solidity version 0.4.26 (including this version).
- *Cause*: Before *Solidity* version 0.5.0, using the *do-while-statements* can cause a bug. Executing the a *continue-statement* in the *do-while-statements* causes the condition determination part to be skipped once.

D. ***Interaction***

D-a. Contract call

(1) Re-entrancy vulnerability (D-a-R) [19, 28]
- *Level*: Ethereum protocol/virtual machine.
- *Cause*:In Ethereum, smart contracts can call other contracts. When the *call-statement* is used to call other contracts, the power of control is transferred to the callee, 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, consequently callees can call back callers.
  (b) The amount of *gas* to be carried is not specified. Consequently, the callee can enter the caller multiple times until the *gas* is exhausted.
  (c) No callee's response function is specified, and the callee can use the *fallback* function to respond to the call and call back the caller in the *fallback* function.
  (d) Ethers are transferred first and balance is deduced later. Because re-entrabcy occurs when transferring ethers, the deduction balance statement will not be executed, and the callee can take ethers multiple times.
- *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) [20]
- *Level*: Ethereum protocol/virtual machine.
- *Cause*: In Ethereum, if a contract uses *call-statements*, *delegatecall-statements*, or *send-statments* to interact with another address and then an exception occurs, the transaction is not terminated, only the result is notified with the return *false*.

D-b. Ether flow

(1) Forced to receive Ether (D-b-F) [30]
- *Level*: Ethereum protocol/virtual machine.
- *Cause*: In Ethereum, an attacker can force ethers to be sent to an address through self-destructing contracts or mining, even if the contract does not want to receive ethers.

(2) Locked Ether (D-b-L) [22]
- *Level*: Developer mistakes.
- *Cause*: In Ethereum, contracts can receive or send ethers. If a contract wants to receive ethers, at least one function of the contract needs to be declared *payable*. At the same time, at least one statement in the contract should be used to transfer out the ethers of the contract, otherwise, all the ethers of the contract will be locked.

(3) Pre-sent ether (D-b-P) [31]
- *Level*: Ethereum protocol/virtual machine.
- *Cause*: 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 ether, then the *present ethers* can interfere with the function of the contract.

E. ***Interface***

E-a. Parameter

(1) Call/delegatecall data/address is controlled externally (E-a-C) [16, 34]
- *Level*: Ethereum protocol/virtual machine.
- *Cause*: *Solidity* allows the smart contract to call other contracts, and the call address and data can be controlled by the developer or external caller. If the call data or call address is controlled externally, the attacker can arbitrarily specify the call address (which can be the malicious contract the attacker developed), call function and parameters (which can be the malicious function and appropriate parameters in the malicious contract), which is very dangerous.

(2) Hash Collisions With Multiple Variable Length Arguments (E-a-H) [30]
- *Level*: Ethereum protocol/virtual machine.
- *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. In some cases, using *abi.encodePacked()* with multiple variable-length parameters can cause hash collisions.

(3) Short address attack (E-a-SA) [8]
- *Level*: Ethereum protocol/virtual machine.
- *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. This can be used to launch *short address attack*. 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 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.

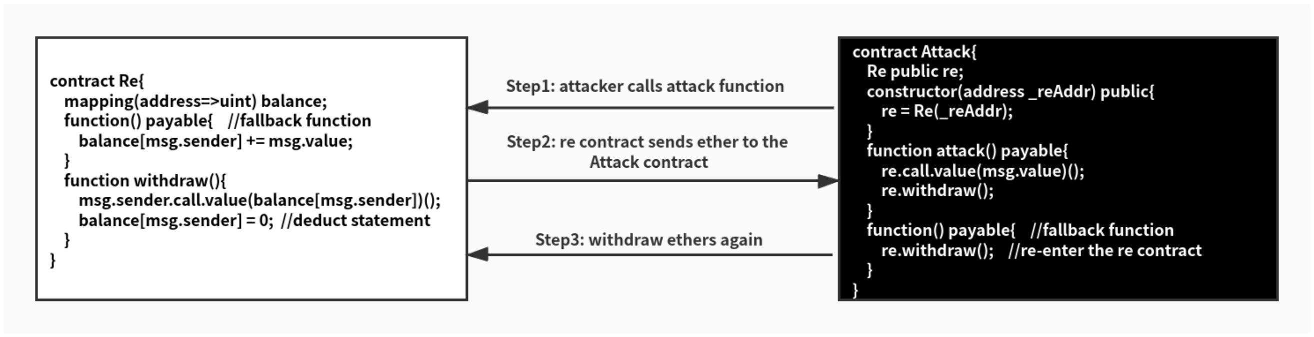
(4) Same function signature (E-a-SF) [31]
- *Level*: Ethereum protocol/virtual machine.
- *Cause*: The function signature (function selector) refers to the first four bytes of the call data for a function call. it specifies the function to be called. The function signature is generated by the *keccak256()*. In some cases, two different functions may have the same function signature.

(5) Signature with wrong parameter (E-a-SW) [27]
- *Level*: Developer mistakes.
- *Cause*: *keccak256()* and *ecrecover()* are built-in functions. *keccak256()* calculates the signature of the public key, while *ecrecover()* uses the signature to recover the public key. 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.

E-b. Token Interface

(1) Nonstandard token interface (E-b-T) [10, 35]
- *Level*: Developer misktakes.
- *Cause*: In Ethereum, *ERC20*, *ERC721* and other token standard specify the functions, events and other information that should be included in the token contracts. Following these standards to develop a token contract enables the contract to interact with other contracts normally.

F. ***Logic***

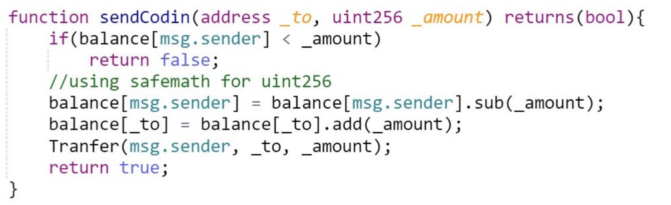


```
contract Re{
   mapping(address=>uint) balance;
   function() payable{   //fallback function
      balance[msg.sender] += msg.value;
   }
   function withdraw(){
      msg.sender.call.value(balance[msg.sender])();
      balance[msg.sender] = 0; //deduct statement
   }
}
```

```
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
   }
}
```

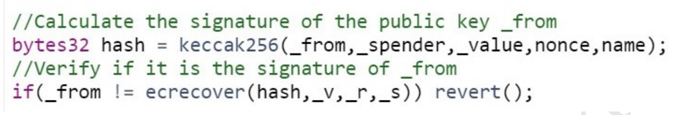
Step1: attacker calls attack function
Step2: re contract sends ether to the Attack contract
Step3: withdraw ethers again

Figure 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;
}
```

Figure 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();
```

Figure 3: Use *keccak256()* and *ecrecover()* to verify identity

F-a. Assembly code

(1) Returning results using assembly code in the constructor (F-a-R) [32]
- *Level*: Ethereum protocol/virtual machine.
- *Cause*: The contract was not fully formed when the constructor was executed. 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) [30]
- *Level*: Ethereum protocol/virtual machine.
- *Cause*: Using assembly code gives developers more flexibility in developing smart contracts, which brings flexibility but is more dangerous. A known danger is that function variables can be specified as any type through assembly code, which can be dangerous.

F-b. Denial of service

(1) DOS by complex fallback function (F-b-DBC) [16]
- *Level*: Ethereum protocol/virtual machine.
- *Cause*: In Ethereum, the contracts always use the *fallback* function by default to respond to transfers into ethers. Generally, most contracts use *send-statement* or *transfer-statement* to send ethers, both of which carry only 2300 *gas*.

(2) DOS by gaslimit (F-b-DBG) [9, 32]
- *Level*: Blockchain.
- *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*, even if the user pays enough *gas*, the transaction may be refused to be packaged.

(3) DOS by non-existent address or abnormal contract (F-b-DBN) [32]
- *Level*: Ethereum protocol/virtual machine.
- *Cause*: When the address that interacts with the contract does not exist, or the callee contract has an exception, the call will fail. To prevent the features of the contract from being affected by this bug, the following principles should be followed:
  (a) The features of the contract should not be limited by a particular key address. For insurance purposes, an alternate address can usually be set.
  (b) When transferring ethers, do not use *transfer-statement* in a loop, and try to let users withdraw ethers instead of remittance.

F-c. Fairness

(1) Results of contract execution affected by miners (F-c-R) [16, 20]
- *Level*: Blockchain.
- *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) [34]
- *Level*: Blockchain.
- *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) [34]
- *Level*: Ethereum protocol/virtual machine.

- *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*

G-a. Gas

(1) byte[] (G-a-B) [29]
- *Level*: Developer mistakes.
- *Cause*: The type *byte[]* is an array of bytes, but due to padding rules, it wastes 31 *bytes* of space for each element (except in storage). It is better to use the *bytes* type instead.

(2) Invariants in loop (G-a-II) [32]
- *Level*: Developer mistakes.
- *Cause*: Placing invariants in loops causes extra *gas* consumption.

(3) Invariant state variables are not declared constant (G-a-IS) [12]
- *Level*: Developer mistakes.
- *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) [12]
- *Level*: Developer mistakes.
- *Cause*: In *Solidity*, 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*

H-a. Authority control

(1) Replay attack (H-a-R) [30]
- *Level*: Ethereum protocol/virtual machine.
- *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.

(2) Suicide contracts (H-a-S) [22]
- *Level*: Ethereum protocol/virtual machine.
- *Cause*: Smart contracts are allowed to *self-destruct* in Ethereum, and the balance of self-destructing contracts will be sent to the specified address. 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) [38]
- *Level*: Ethereum protocol/virtual machine.
- *Cause*: *Solidity* provides the keyword *tx.origin* to indicate the originator of the transaction. Using *tx.origin* for authentication can cause an attacker to bypass authentication. 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) [22]
- *Level*: Ethereum protocol/virtual machine.
- *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, thereby allowing anyone to withdraw ethers from the contract. In some cases, this bug was caused by a typo in the constructor name.

(5) Wrong constructor name (H-a-WCN) [32]
- *Level*: Ethereum protocol/virtual machine.
- *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. This puts the contract in danger, as the constructor is usually responsible for assigning values to key state variables. The difference between this kind of bug and *Wasteful contracts* bug is that this kind of bug may cause *Wasteful contracts* bug, but not all *Wasteful contracts* bugs are caused by this bug. In some cases, this kind of bug will also cause other problems.

H-b. Privacy

(1) Non-private variables are accessed by public/external functions (H-b-NV) [26]
- *Level*: Developer mistakes.
- *Cause*: *Solidity* language requires visibility of specified state variables, of which *internal* and *private* specify that state variables can only be accessed internally. But developers can still use the *public* or *external* functions to access the *internal* and *private* state variables, which may lead to accidental exposure of privacy.

(2) Public data (H-b-PD) [38]
- *Level*: Blockchain.
- *Cause*: Because Ethereum is based on the blockchain, the miner has complete network data backup. For miners, all contract codes and the values of the variables are visible, even if external visibility is specified using *private*.

I. *Standard*

I-a. Maintainability

(1) Implicit visibility level (I-a-I) [18]
- *Level*: Developer mistakes.
- *Cause*: Before *Solidity* version 0.5.0, if developers do not explicitly specify the visibility of state variables and functions, *Solidity* sets the visibility of state variables and functions to default values. Failure to explicitly specify visibility can make the code difficult to understand, and state variables that do not explicitly specify visibility will be set to *private* visibility.

(2) Nonstandard naming (I-a-N) [29]
- *Level*: Developer mistakes.
- *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) [12]
- *Level*: Developer mistakes.

  - *Cause*: Writing many digits makes the code difficult to read and review, and increases the probability of making mistakes.

(4) Unlimited compiler versions (I-a-UC) [9]
  - *Level*: Developer mistakes.
  - *Cause*: *Solidity* is a rapidly evolving language. In different versions of *Solidity*, different statements may have different semantics. When writing contracts, the language version should be explicitly specified (a fixed version or limited interval) .

(5) Use deprecated built-in symbols (I-a-UD) [32]
  - *Level*: Smart contract programming language.
  - *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) [32]
  - *Level*: Smart contract programming language.
  - *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, which means that developers can modify data in functions declared as *view* or *constant*.

(2) Improper use of *require*, *assert*, and *revert* (I-b-I) [30]
  - *Level*: Developer mistakes.
  - *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. For example, *require* should be used for input validation, *assert* should be used to validate invariants, and *revert* is used for termination and rollback transactions. In addition, the *require* does not consume any *gas*, but *assert* consumes all available *gas*.

3.5 Severity grading of smart contract bugs

To give developers and researchers a clear understanding of the consequence of each bug, we will grade the severity of each bug. According to the *IEEE Standard Classification for Software Anomalies[14]*, 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 3.5. bularcc|cc

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-SF	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

Table 1: Severity level distribution of the number of bugs

Critical	Major	Minor	Inconsequential
4	25	16	5

Table 2: Comparison of *Jiuzhou* with other data sets

Data set	Number of contracts	Kinds of bugs covered	*Solidity version*
Jiuzhou	174	50	0.4.24 to 0.6.4
ethernaut[12]	21	21	0.4.18 to 0.4.24
not-so-smart-contracts[13]	25	12	0.4.9 to 0.4.23
SWC-registry[14]	114	33	0.4.0 to 0.5.0

I-a-N	Inconsequential	I-a-T	Major
I-a-UC	Inconsequential	I-a-UD	Inconsequential
I-b-I	Minor	I-b-F	Minor

Table 1 shows the distribution of the number of bugs in each severity level.

4 *JIUZHOU*: A DATA SET FOR SMART CONTRACT BUGS

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. The dataset is available at: https://github.com/xf97/JiuZhou.

Jiuzhou provides 174 (to do) smart contracts, which are developed in *Solidity* language. All contracts are written based on the characteristics of the bugs. Blockchain, Ethereum protocol / virtual machine, and developer mistakes, the above three levels of bug smart contracts are developed using the latest *Solidity* language version (0.6.4). The smart contract programming language level bug smart contracts use the last *Solidity* version with this kind of bug. Besides the smart contract source files, *Jiuzhou* also provides compiled bytecode files. Compared with several commonly used smart contract datasets [23, 24, 30], *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 2.

5 EXPERIMENT

5.1 Experimental design

We designed a set of experiments to measure the ability of each data set to evaluate smart contract analysis tools.

[12]https://ethernaut.openzeppelin.com
[13]https://github.com/crytic/not-so-smart-contracts
[14]https://github.com/SmartContractSecurity/SWC-registry

First of all, by issuing questionnaires, we obtained the knowledge of smart contract developers on the detection capabilities of different smart contract analysis tools.

Then, we use different smart contract analysis tools to detect different data to obtain the detection performance of different smart contract analysis tools in different data sets.

Finally, by comparing artificial recognition and detection performance, it is concluded that each data set reflects the performance of the smart contract analysis tool's detection capability.

5.2 Questionnaire design

We used *smart contract* as a search keyword, retrieved 639 smart contract developers from *Github [15]*, obtained their mailboxes from their past submission records or personal information, and then sent our questionnaire to their mailboxes. We sent a total of x(to do) questionnaires and received x (to do) responses.

5.3 Detection ability experiment

5.4 Analysis of results

6 RELATED WORK

6.1 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. [6] 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. [5] summarize four common smart contract programming pitfalls by investigating the mistakes students when they are teaching smart contract programming. Atezi et al. [1] summarize 11 kinds of programming traps that may lead to security bugs. They believe that one of the main reasons for the continuous proliferation of smart contract bugs is the lack of inductive documentation for smart contract bugs. Dingman et al. [7] 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. [3] 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. [36] 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. [39] 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.

6.2 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. [20] 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. [17] 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. [16] 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. [38] 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. [4] 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.

7 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 based on their causes. 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 plan to provide misleading contracts that may cause smart contract analysis tools to make mistakes in order to better compare existing tools. Second, we will focus on developing an Ethereum smart contract analysis tool that combines the advantages of source-code and bytecode to cover the statistical bugs in the paper.

REFERENCES

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*. Springer, 164–186.

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

[3] Jiachi Chen, Xin Xia, David Lo, John Grundy, Daniel Xiapu Luo, and Ting Chen. 2019. Domain Specific Code Smells in Smart Contracts. *arXiv preprint arXiv:1905.01467* (2019).

[4] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 442–446.

[5] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. 2016. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*. Springer, 79–94.

[6] Giuseppe Destefanis, Michele Marchesi, Marco Ortu, Roberto Tonelli, Andrea Bracciali, and Robert Hierons. 2018. Smart contracts vulnerabilities: a call for blockchain software engineering?. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 19–25.

[7] W. Dingman, A. Cohen, N. Ferrara, A. Lynch, P. Jasinski, P. E. Black, and L. Deng. 2019. Classification of Smart Contract Bugs Using the NIST Bugs Framework. In *2019 IEEE 17th International Conference on Software Engineering Research, Management and Applications (SERA)*. 116–123. https://doi.org/10.1109/SERA.2019.8886793

[8] Doingblock. [n. d.]. smart contract security. https://github.com/doingblock/smart-contract-security. Accessed Mar, 11, 2020.

[9] *Ethereum*. [n. d.]. Browser-Only Ethereum IDE and Runtime Environment. https://remix.ethereum.org/. Accessed Jan 7,2020.

[10] William Entriken. [n. d.]. EIP 721: ERC-721 Token Standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md. Accessed Jan 7,2020.

[11] Martin Paul Eve. 2007. Right-To-Left and Left-To-Right characters. *martineve.com* (2007).

[12] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.

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

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

[15] GitHub Inc. [n. d.]. Open-source project repository. https://github.com/. Accessed Dec 19,2019.

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

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

[18] knownsec. [n. d.]. Ethereum Smart Contracts Security CheckList From Knownsec 404 Team. https://github.com/knownsec/Ethereum-Smart-Contracts-Security-CheckList. Accessed Jan 7,2020.

[19] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 65–68.

[20] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 254–269.

[21] Satoshi Nakamoto. 2019. *Bitcoin: A peer-to-peer electronic cash system*. Technical Report. Manubot.

[22] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 653–663.

[23] Trail of Bits. [n. d.]. Vulnerable smart contracts. https://github.com/crytic/not-so-smart-contracts. Accessed Jan 7,2020.

[24] OpenZeppelin. [n. d.]. Web3/Solidity based wargame. https://ethernaut.openzeppelin.com/. Accessed Jan 7,2020.

[25] Reza M Parizi, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Amritraj Singh. 2018. 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., 103–113.

[26] quantstamp. [n. d.]. solidity-analyzer. https://github.com/quantstamp/solidity-analyzer. Accessed Mar, 12, 2020.

[27] sec bit. [n. d.]. awesome-buggy-erc20-tokens. https://github.com/sec-bit/awesome-buggy-erc20-tokens/blob/master/ERC20$_token_issue_list.md. AccessedMar$, 11, 2020.

[28] David Siegel. 2016. Understanding the dao attack. *Retrieved June* 13 (2016), 2018.

[29] SmartContractSecurity. [n. d.]. The development documents of *Solidity*. https://solidity.readthedocs.io/en/v0.6.1/. Accessed Jan 7,2020.

[30] SmartContractSecurity. [n. d.]. Smart Contract Weakness Classification and Test Cases. https://swcregistry.io/. Accessed Jan 7,2020.

[31] smartdec. [n. d.]. classification. https://github.com/smartdec/classification. Accessed Mar, 11, 2020.

[32] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 9–16.

[33] Christof Ferreira Torres, Julian Schütte, et al. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 664–676.

[34] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 67–82.
[35] Fabian Vogelsteller. [n. d.]. EIP 20: ERC-20 Token Standard. https://eips.ethereum.org/EIPS/eip-20/. Accessed Jan 7,2020.
[36] Shuai Wang, Liwei Ouyang, Yong Yuan, Xiaochun Ni, Xuan Han, and Fei-Yue Wang. 2019. Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* (2019).
[37] Maximilian Wohrer and Uwe Zdun. 2018. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2–8.
[38] Pengcheng Zhang, Feng Xiao, and Xiapu Luo. 2019. SolidityCheck: Quickly Detecting Smart Contract Problems Through Regular Expressions. *arXiv preprint arXiv:1911.09425* (2019).
[39] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach D Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2019. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering* (2019).

A BUG CORRESPONDENCE TABLE BEFORE AND AFTER MERGING

In Appendix A, we use Table **??** to show the correspondence of bugs before and after the merger and indicate the source of the bugs in this paper.

B QUESTIONNAIRE