Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum

JIACHI CHEN, Monash University
XIN XIA, Monash University
DAVID LO, the School of Information Systems, Singapore Management University
JOHN GRUNDY, Monash University

The *Selfdestruct* function is provided by Ethereum smart contracts to destroy a contract on the blockchain system. However, it is a double-edged sword for developers. On the one hand, using *Selfdestruct* function enables developers to remove smart contracts (SC) from Ethereum and transfers Ethers when emergency situations happen, e.g. being attacked. On the other hand, this function can increase the complexity for the development and open an attack vector for attackers. To better understand the reasons why SC developers include or exclude the *Selfdestruct* function in their contracts, we conducted an online survey to collect feedback from them and summarize the key reasons. Their feedback shows that 66.67% of the developers will deploy an updated contract to the Ethereum after destructing the old contract. According to this information, we propose a method to find the self-destruct contracts (also called predecessor contracts) and their updated version (successor contracts) by computing the code similarity. By analyzing the difference between the predecessor contracts and their successor contracts, we found six reasons that led to the death of the contracts; three of them (i.e., *Unmatched ERC20 Token, Confusing Contract and Limits of Permission*) might affect the life span of contracts. We developed a tool named LifeScope to detect these problems. LifeScope reports 0 false positives or negatives in detecting *Unmatched ERC20 Token* and *Confusing Contract*. In terms of *Limits of Permission*, LifeScope achieves 85.67% of F-measure and 0.9014 of AUC. According to the feedback of developers who exclude *Selfdestruct* functions, we propose suggestions to help developers use *Selfdestruct* functions in Ethereum smart contracts better.

Additional Key Words and Phrases: Smart Contract, Ethereum, Selfdestruct Function, Empirical Study

ACM Reference format:
Jiachi Chen, Xin Xia, David Lo, and John Grundy. 2016. Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum. 1, 1, Article 1 (January 2016), 27 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The great success of Bitcoin [41] shows the enormous potential of blockchain technology [9]. People usually regard Bitcoin as a representative of blockchain 1.0 [39], the first generation of blockchain technology. In blockchain 1.0, the blockchain technology is usually used to make cryptocurrency [10], e.g., Bitcoin, Ripple Coin [6]. The usage scenario of cryptocurrencies in blockchain 1.0 is limited, as the main application for them is storing and transferring values [27].

The birth of Ethereum [11] changed this situation at the end of 2015. Ethereum leverages a technology named *smart contracts*, which can be regarded as a program that runs on the blockchain.

Smart contracts are usually developed in a high-level programming language, e.g., Solidity [14]. The blockchain technology provides an immutability feature for smart contracts, which means all of the smart contracts are self-executed and can't be modified. Even the creator of the contract cannot modify the code after deploying the contract to the Ethereum. By utilizing a smart contract, developers can easily design their DApps (decentralized applications) [2]. The appearance of Ethereum marked the point that blockchain technology upgraded from blockchain 1.0 to blockchain 2.0. By Sept. 2019, millions of smart contracts on the Ethereum [12] have been applied to different fields, such as gaming [7] and monetization [3], with many other application domains under exploration.

However, the features of Ethereum also make it easy to be attacked. First, Ethereum is a permission-less network; smart contracts on Ethereum can be executed by everyone, including attackers. Second, all the data stored on the blockchain, transactions, and bytecode of smart contracts are visible to the public, which makes smart contracts become attractive targets for attackers. In 2016, attackers utilized a vulnerability (reentrancy [40]) to attack a smart contract owned by an organization named DAO (Decentralized Autonomous Organization). This attack made the organization lose 3.6 million Ethers[1]. People usually call this attack a DAO attack [1]. Actually, the attack continued for several days and the organization even noticed that their contract had been attacked at that time. However, they could not stop the attack or transfer the Ethers because of the immutability feature of smart contracts.

This DAO attack attracted great attention from both academia and industry. Some previous works [4, 31] advice contracts add some mechanisms to stop the contracts or transfer the Ethers when emergency situations happen, e.g., a contract being attacked. In this case, the owners can reduce the impact of financial loss. Solidity provides a novel *Selfdestruct* function [14]. By calling this *Selfdestruct* function, a smart contract can be removed from the blockchain and all the Ethers on the contract will be transfered to a specified address, which is an unique identification for the account[2] on Ethereum.

This *Selfdestruct* function is however a double-edged sword for developers. On the one hand, the function enables contract owners have the ability to reduce financial loss when emergency situations happen. On the other hand, this function is also harmful. The function might open an attack vector for attackers. It may also lead to a trust concern from the contract users, as the contract owners can transfer user's Ethers that are stored on the contract. These conflicting features make the *Selfdestruct* function valuable to be investigated. **In this paper, we call a smart contract that has been self-destructed by calling *Selfdestruct* function as a 'self-destructed contract'.** To better understand the developers' perspective about this unique Ethereum smart contract *Selfdestruct* function, we designed an online survey to collect their opinions, and to help us to answer the following research question:

RQ1: Why do smart contract developers include or exclude Selfdestruct functions in their contracts?

We sent our survey to 996 smart contract developers and received 88 responses. Their feedback shows that there are six reasons why developers exclude the *Selfdestruct* function. The top two most popular reasons are *security concerns* and *trust concerns*. Developers are worried that the *Selfdestruct* function in their contract will open an attack vector for attackers. Besides, this function can also reduce users' confidence of the contract as the contract owner have ability to transfer users' Ethers that stored on the contract balance. To address these concerns of developers, we provide

[1]Ether is the cryptocurrency generated by the Ethereum platform. An Ether worth $270 on Feb. 2020.
[2]There are two types of accounts on Ethereum, i.e., External Owned Account (EOA) and contract account. EOA is controlled by users. Contract account is controlled by its code.

five suggestions in Section 6, which can help developers to better use the *Selfdestruct* function. In terms of why developers include a *Selfdestruct* function, our survey feedback shows that two thirds of developers will kill their contracts when security vulnerabilities are found, or if they want to upgrade their smart contract's functionalities. After fixing bugs or upgrading the contracts, they will deploy a new version of the contract. This finding inspired our second research question:

RQ2: Why do smart contracts on Ethereum self-destruct?

In answering RQ1, we found that two thirds of developers will deploy a new version of the contract after the old contract executed *Selfdestruct* function. We call a smart contract whose *Selfdestruct* function has been executed a *'self-destruct'* contract. If the *self-destruct* contract can find its new version of the contract, the new contract is the *'Successor'* contract, and the self-destruct contract is the *'Predecessor'* contract of the *successor* contract. By comparing the difference between *Predecessor* contract and *Successor* contract, we can identify the reasons why contract destructed, e.g., security reasons. We propose a method that leverages a clone detection tool (SmartEmbed [33, 34]) to find *Predecessor* contracts and their *Successor* contracts. Then, we summarize 6 common reasons why contracts destructed by conducting *open card sorting* [44].

As a result, we summarize 6 common self-destruct reasons, detailed in Table 1. Three of them – *Unmatched ERC20 Token, Confusing Contract* and *Limits of Permission* – might affect the life span of contracts. Therefore, an automatic tool to detect these problems would be helpful to extend the life span of smart contracts. This motivated us to investigate our third research question:

RQ3: How can we detect lifespan-based smart contract problems automatically?

We designed a tool named LifeScope, which can be used to detect *Confusing Contract, Unmatched ERC20 Token*, and *Limits of Authority* problems. For the first two problems, LifeScope uses ASTs (Abstract Syntax Trees) to parse the source code and extract related information. LifeScope obtains 100% of F-measure for detecting these two problems. For *Limits of Authority*, LifeScope first transfers code to a TF-IDF representation and utilizes a machine learning method to predict the permission. LifeScope achieves an F-measure and AUC of 85.67% and 0.9014 for this task.

The main contributions of this paper are:

- To the best of our knowledge, this is the first empirical work that investigates the *Selfdestruct* function of smart contracts in Ethereum. We conduct an online survey to collect feedback from developers. According to this survey feedback, we summarize 5 reasons why developers add *Selfdestruct* functions and 6 reasons why they do not add them to their smart contracts.
- We design an approach to find six reasons why smart contracts destructed. These self-destruct reasons can be used as a guidance when practitioners develop their contracts. Also, our approach gives inspiration for researchers. They can use the same approach to find more self-destruct reasons and apply the method to other smart contract platforms, e.g., EOS[3] [8].
- We propose a tool named LifeScope to detect three problems that might shorten the life span of smart contracts. LifeScope obtains 100% of F-measure in detecting *Confusing Contract* and *Unmatched ERC20 Token*. And it achieves an F-measure and AUC of 85.67% and 0.9014, respectively in detecting *Limits of Authority*.
- According to the feedback from our survey, there are six common reasons why some developers do not use *Selfdestruct* function. We give five suggestions for developers to address these issues and to help them better use the*Selfdestruct* function in their smart contacts.

[3]EOS is another popular blockchain platform which support the running of smart contracts launched in mid-2018.

The organization of the rest of this paper is as follows. In Section 2, we present the background knowledge of smart contracts. Then, we show the answer to the three research questions in Section 3-5, respectively. We discuss the implication, how to better utilize *Selfdestruct* function and threats to validity in Section 6. After that, we introduce related works in Section 7. In Section 8, we conclude the whole work and present our future work.

2 BACKGROUND

In this section, we briefly introduce the background information about smart contracts, the Ethereum system, and some features and knowledge about smart contract programming.

2.1 Smart Contracts

Bitcoin was the first cryptocurrency that utilized blockchain as its underlying technology. It allows users to encode scripts to process transactions. However, the scripts on Bitcoin are not Turing-complete, which restricts the usage of Bitcoin. In contrast, Ethereum leverages a technology named *smart contracts*. These can be regarded as self-executed programs that run on the blockchain. When developers deploy smart contracts to Ethereum, the source code of the contracts will be compiled into bytecode and reside on the blockchain forever. The storage of Ethereum is very expensive, as all the data stored on the blockchain will be copied on each node, a so-called distributed ledger. To minimize the data space, the source code of the smart contracts will not be stored on the blockchain. Once a contract is deployed to the blockchain, the contract is identified by a 20-byte hexadecimal address. Arbitrary users can call the functions of a smart contract by sending transactions to the contract address.

```
pragma solidity ^0.4.25;
contract Example{
  address owner_addr;
  address[] participators;
  uint participatorID = 0;
  function constructor(){
    owner_addr = msg.sender;
  }
  function() payable{
    if(msg.value != 1 Ether)
      revert();
    participators[participatorID] = msg.sender;
    participatorID++;
    if(this.balance == 10 Ether)
        getWinner();
  }
  function getWinner(){
    uint random = uint(block.blockhash(block.number)) % participants.length;
    participators[random].transfer(9 Ether);
    participatorID = 0;
  }
  modifier onlyOwner{
    if(msg.sender != owner_addr)
      _;
  }
  function Selfdestructs(address addr) onlyOwner(){
    selfdestruct(addr);
  }
}
```

Listing 1. A simple contract

Listing 1 is an example of a smart contract that implements a simple gambling game by using Solidity [14]. Solidity is the most popular smart contract programming language on the Ethereum platform. Users can send 1 Ether to the contract. Once the contract receives 10 Ethers, the contract will choose 1 user as the winner randomly and send 9 Ethers to him/her.

The first line is called the version pragma, which is used to identify the compiler version of the contract. Lines 3-5 are the global parameters, and the function on line 6 is the constructor function of the smart contract. The constructor function can only be executed once when deploying the contract to the blockchain. Therefore, this function is usually used to store the owner's information. Specifically, Line 7 stores the owner's address by using *msg.sender*. (*msg.sender* is used to obtain the address of the transaction sender.) Function on line 9 named fallback function, which is the only unnamed function of the smart contract. This function will be executed automatically when an error function call happens. For example, a user calls function "δ", but there is no function named "δ" in the contract. In this situation, a fallback function will be executed to handle the error call. If the fallback function is marked by a keyword named *payable*, the fallback function will also be executed automatically when the contract receives Ethers. Lines 9 and 10 guarantee that each user sends 1 Ether to the contract. If the user sends other amounts of Ethers, the transaction will be rolled backed by executing *revert()*, which is a function provided by *Solidity*. When the contract receives 10 Ethers (Line 14), the contract will choose 1 user to send 9 Ethers by using function *getWinner* (Line 17). The contract generates a random number by using the block info related functions[4] in Line 18. Then, the contract sends 9 Ether to the winner in Line 19.

2.2 Function Modifier

Ethereum is a permission-less network – everyone can call methods to execute smart contracts. Developers usually add permission checks for permission-sensitive functions. For example, the contract in Listing 1 records the owner's address in its constructor function (Line 7). In this case, the contract can compare whether the caller's address is the same as the owner's address. Solidity provides *Function Modifiers* which are used to add prerequisites checks to a function call. A function with function modifier can be executed only if it passes the check of the modifier.

Listing 1 line 22 shows a modifier named *onlyOwner*. This modifier requires the transaction creator(*msg.sender*) should be the owner of the contracts (*owner_addr*). Function *Selfdestructs* on line 26 contains this modifier. Therefore, only the owner of the contract can call *Selfdestruct* function in line 27.

2.3 Selfdestruct Function

Selfdestruct function in Listing 1 line 27 is the only way to remove the contract from Ethereum. When executing this method, the caller can transfer all Ethers on balance to a specific address (*addr*) (line 27). Then, the contract address will be discarded. If others transfer Ethers to the self-destructed contract address, the Ethers will be locked forever.

This function is sometimes harmful as the immutability feature can be broken. Immutability is a special and important feature of smart contracts compared to traditional programs. Once a contract is deployed to the blockchain, none can modify the contract, even the owner. However, this function can allow the owner to kill the contract and make the contract disappear from the

[4] (block.blockhash and block.number are the functions provide by Solidity to obtain block related information. Since block hash number is random; so it can be used to generate random numbers sometimes.)

blockchain. This might reduce the confidence of the users, as the owner can transfer all the Ethers of the contract. For example, the owner can transfer all the Ethers by calling the *Selfdestruct* function on contract in Listing 1 when the contract receives 9 Ethers. In this case, all the users are losers.

```
1  contract Victim {
2      mapping(address => uint) public userBalannce;
3      function withDraw(){
4          uint amount = userBalannce[msg.sender];
5          if(amount > 0){
6              msg.sender.call.value(amount)();
7              userBalannce[msg.sender] = 0;
8          }
9      }
10         ...
11 }
12 contract Attacker{
13     function() payable{
14         Victim(msg.sender).withDraw();
15     }
16     function reentrancy(address addr){
17         Victim(addr).withDraw();
18     }
19         ...
20 }
```

Listing 2. The Demo of the DAO Attack

2.4 The DAO Attack - A Motivation Example of *Selfdestruct* Function

In 2016, attackers found a vulnerability named Reentrancy [31, 40] in a smart contract of the Decentralized Autonomous Organization (DAO organization), and this vulnerability made the DAO organization lost 3.6 million Ethers ($270/Ether on Feb. 2020). People usually call this infamous attack a DAO attack.

List 2 is a demo of the DAO attack. There are two smart contracts, i.e., *Victim* contract and *Attacker* contract. The *Attacker* contract is used to transfer Ethers from *Victim* contract, and the *Victim* contract can be regarded as a bank, which stores the Ethers of users. Users can withdraw their Ethers by invoking *withDraw()* function. However, *withDraw()* function contains the *Reentrancy* vulnerability in line 6-7.

First, the *Attacker* contract uses *reentrancy()* function (L16) to invokes *Victim* contractfis *withDraw()* function in line 3. The *addr* in line 17 is the address of the *Victim* contract. Normally, the *Victim* contract sends Ethers to the callee in line 6, and resets callee's balance to 0 in line 7. However, Ethereum does not support concurrency, which means *Victim* contract sends Ethers to Attacker contract before resetting the balance to 0. When the *Victim* contract sends Ethers to the *Attacker* contract, the fallback function (L13) of the *Attacker* contract will be invoked automatically, and Line 7 is not executed at that time. So, the *Attacker* contract can invoke *withDraw()* function repeatably.

Actually, the DAO attack continued for several days and the organization even noticed that their contract had been attacked at that time. However, they could not stop the attack or transfer the Ethers because of the immutability feature of smart contracts. If the contract contains a *selfdestruct* function, the DAO organization can transfer all the Ethers easily, and reduce the financial loss.

2.5 Predecessor Contracts and Successor Contracts

Using the *Selfdestruct* function is the only way to remove code from the blockchain. After executing this function, the contract will be destroyed, and there is no way to recover it. In this paper, we call a smart contract that has executed a *Selfdestruct* function a *'self-destruct'* contract. There are many reasons why contracts might be designed to self-destruct, e.g., to mitigate a later found security vulnerability or to enable a later functional change. Many developers will upgrade the *self-destruct contract* and deploy a new version. We call the new version of the contract a *'Successor'* Contract, and the self-destructed contract is the *Predecessor* contract of the *Successor* contract.

2.6 ERC20 Standard

Motivated by the great success of Bitcoin, thousands of cryptocurrencies have been created in recent years. However, most of them do not have their own blockchain system. Instead, they are usually implemented by smart contracts that run on the Ethereum, also called *tokens*. To ensure different tokens can interact accurately and be reused by other applications (e.g., wallets and exchange markets), Ethereum provides ways to standardize their behaviors. ERC20 [3] is the most popular token standard on Ethereum. It defines 9 standard interfaces (3 are optional) and 2 standard events. To design ERC20 tokens, developers should strictly follow the standard. For example, the standard method TRANSFER is declared as "function transfer(address _to, uint256 _value) public returns (bool success)". This function is used for transferring tokens to a specific address (_to). The ERC20 standard requires this function to throw an exception if the caller's account balance does not have enough tokens to spend. Besides, the function should fire an event named "TRANSFER" to inform the caller whether the tokens are transferred successfully.

2.7 Etherscan and Verified Smart Contracts

Although blockchain-based systems like Ethereum store almost all of the data, e.g., transaction information, users' balance, it is not easy to search data on Ethereum directly. To facilitate data search of Ethereum, Etherscan [12] was developed to be a blockchain explorer for Ethereum. By using Etherscan, users can quickly search the Ethereum blockchain for transactions, addresses, tokens, prices and other activities taking place on Ethereum. Etherscan also provides many other functionalities which are beneficial to the development of the Ethereum ecosystem. For example, Ethereum only stores the bytecode of smart contracts to reduce the data size stored on the blockchain. However, source code is important for code reuse and the development of the Ethereum ecosystem. To address this limitation of the Ethereum, developers can upload the source code of smart contracts on Etherscan. Etherscan uses a *Source code verification* system to verify the source code.

To upload the contracts, developers need to specify the contract address that they want to verify, choose a compiler version of the contract, and give the contract name. There might be several subcontracts in a contract address, but Ethereum only allows one main contract. The contract name is the same as the main contract. For each execution, EVM (Ethereum Virtual Machine) will choose the main contract as entry and execute the contract. The *Source code verification* system of Etherscan will compile the provided smart contract to bytecode and compare whether the compiled bytecode is same as the bytecode in the blockchain. After passing the verification process, the source code can be found at Etherscan. However, EVM will remove unused code when compiling the source code to bytecode. Therefore, the verification system of Etherscan cannot recognize the unused subcontracts.

3 RQ1: DEVELOPER'S PERSPECTIVE ABOUT SELFDESTRUCT FUNCTION

3.1 Motivation

Using the *Selfdestruct* function can enable developers to destruct their contracts and transfer Ethers when emergency situations happen, e.g. a contract is being attacked or is found to be buggy. However, this function is also harmful for both contract users and contract owners. In our analysis, we crawled all of the 54,739 verified smart contracts from Etherscan [12] by the time of writing (for details see Section 4.2.1), and found 2,786 (5.1%) smart contracts contain a *Selfdestruct* function in their source code. In this RQ, we aim to investigate the developers' perspective about using the *Selfdestruct* function in Ethereum smart contracts. By understanding the reason why they include or exclude *Selfdestruct* functions in their contracts, we can better understand the advantages and disadvantages of this function. We then want to design some guidance about using the *Selfdestruct* function, which enables developers design a more robust smart contract.

3.2 Approach

3.2.1 Validation Survey. In this paper, we utilize the methods proposed by Kitchenham et al. [38] to design a survey for collecting the opinions from smart contract developers. To increase the response rate, we make the survey anonymous [46] and provided a raffle for developers who take part in our survey. Participation in the raffle is voluntary; we chose two respondents who provided their email addresses as the winner, and gave them $50 Amazon gift cards as the reward. We first use a small scale survey to collect feedback about our survey. The feedback includes: (1). Whether the expression about our question is easy to understand. (2). Whether the time to finish the survey is reasonable. After the small scale survey, we refine our questionnaire based on the feedback we collected. Finally, conduct a large scale investigation to collect our data. [5]

3.2.2 Survey Design. To understand the background of the respondents better, we first collect their demographic information. These five questions can help us have an overall understanding of the respondents.

a. Demographics:

- 1. Professional smart contract developer? : Yes / No
- 2. Involved in open source software development? : Yes / No
- 3. Main role in developing smart contract: Testing / Development / Management / Other
- 4. Experience in years (decimals ok)?
- 5. Current country of residence ?

After that, we give a brief example of the *Selfdestruct* function and ask them the following questions about this function. The respondents are required to choose yes / no in the question 6. If they choose yes, they are required to answer question 7; otherwise, they should answer question 8. Both question 7 and 8 contain a textbox. The respondents require to write several sentences to describe their reasons.

b. Questions about Selfdestruct Functions:

- 6. Will you add *Selfdestruct* functions in your future smart contracts? : Yes (Go to Q7) / No (Go to Q8)
- 7. Why do you add *Selfdestruct* functions?
- 8. Why do you not add *Selfdestruct* functions?

[5]ETHICS COMMITTEE APPROVAL

To increase the response rate, we prepare two kinds of survey[6], i.e., English Version and Chinese Version, as Chinese is the most spoken language and English is an international language in the world. The Chinese version survey is carefully translated to ensure the contents between the two versions are the same.

3.2.3 Recruitment of Respondents. To receive sufficient response from different backgrounds, we first sent our questionnaire to our contacts who are working in world-famous blockchain companies or doing related research in academic institutions, e.g., *Ant Financial, The Hong Kong Polytechnic University, NUS, The University of Manchester.* Then, we also collect developers' email addresses on their Github homepage who are contributing to open-sourced blockchain projects. We collected 1,238 email addresses from Github. Due to the scale of the smart contract projects, 1,238 are the numbers of contributors of the top 100 most popular (ranked by stars) smart contract related projects. Therefore, we believe 1,238 is a reasonable number to support the reliability of the data.

3.3 Result

Since some email addresses we collected are illegal or abandoned, we successfully sent our survey to 996 developers, and receive 88 responses from 32 countries (The response rate is 8.84%). The top three countries in which respondents reside are China (29.89%), the USA (8.05%) and the UK (5.57%). Three of the respondents claim that they are not professional smart contract developers and have no experience in developing smart contracts. Therefore, we exclude their responses and use the remaining 85 responses for analysis. The average years of experience in developing smart contracts are 1.96 years for all of our respondents. As the survey was undertaken in Aug. 2019 (about 4 years since Ethereum was first published), 1.96 average years of experience shows that they have good experience in developing smart contracts. Among these respondents, 62 (72.94%), 10 (11.76%), 5 (5.88%) described their job roles as development, testing, and management, respectively. The other 8 respondents said they have multiple roles, such as security auditor and research.

3.3.1 Reasons for including the Selfdestruct function. 33 of the respondents claim that they will add the *Selfdestruct* function in their smart contracts. We analyzed the feedback of these respondents and summarized five key reasons. **As some respondents give more than one reason, the sum of these is higher than 33.**

Reason 1: Security Concerns. **18 respondents** claim that they use the *Selfdestruct* function to stop the contracts when security vulnerabilities are detected in their contracts. After fixing the vulnerabilities, they can deploy a new contract.

Reason 2: Clean Up Environment. Blockchain is a distributed ledger where each node stores all the data. After destroying the contracts, the functions of the contracts cannot be called anymore. **11 respondents** mention that when the duty of the contract is finished, they will call the *Selfdestruct* function to remove the contracts from the blockchain, which can clean up the blockchain environment.

Reason 3: Quickly Withdraw Ethers. By using the *Selfdestruct* function, the owner of the contract can remove all the Ethers to a specific address. **9 respondents** claim this function can help them transfer assets quickly.

[6]The survey can be found at: https://forms.gle/v1N4d8x6syoqoLJXA (English Version) and https://www.wjx.cn/jq/47005808.aspx (Chinese Version).

Reason 4: Upgrade Contracts. **4 respondents** said they may need to upgrade their contracts in the future. Adding a *Selfdestruct* function is the easiest method to upgrade their contract. This function allows them to remove the old version of the contract and deploy a new version.

Reason 5: Business Requirement. The business requirement is also a reason why developers add *Selfdestruct* function. **2 respondents** said their business partners require them to add the *Selfdestruct* function.

According to our survey, 22 out of 33 (66.67%) of the respondents will deploy a new version of the contracts after destructing the old contracts, whether for security concerns or to upgrade the contract. *Selfdestruct* function is useful for contract developers to handle emergency situations, e.g., when serious security issues are found in the contracts.

3.3.2 *Reasons for excluding the Selfdestruct function.*

52 of the respondents claim that they will not add *Selfdestruct* functions in their smart contracts. **As some respondents give more than one reason, the sum of these is higher than 52.**

Reason 1: Security Concerns. The *Selfdestruct* function can also lead to serious security problems if the contract does not handle access permissions correctly. **19 respondents** are worried that the *selfdestruct* function might open an attack vector for adversaries to exploit.

Reason 2: Trust Concerns. **16 respondents** who give this reason believe that immutability is an important feature for smart contracts. Once a contract is deployed to the blockchain, no-one can modify the contract, even the owner. However, including a *Selfdestruct* function might break immutability, which may lead to trust concerns from the contract users. To be specific, *Selfdestruct* function allows the owner to kill the contract and make the contract disappear from the blockchain. Also, the owner can transfer all the Ethers, which raises a trust concern for the user.

Reason 3: Requirement Concerns. **16 respondents** mention that their contracts do not use *Selfdestruct* function as their contracts do not have Ethers. Therefore, they do not need to transfer Ethers. Besides, when they want to add some new functionalities, they said they could deploy a new smart contract and ignore the old one.

Reason 4: Unfamiliarity. **7 respondents** claim that they are unfamiliar with *Selfdestruct* function. They are worried that they might misuse the *Selfdestruct* function, and lead to the bugs.

Reason 5: Additional Complexity. **4 respondents** told us that they need to add more tests if they add *Selfdestruct* functions in the contracts, which can introduce additional complexity to their contracts.

Reason 6: Additional Financial Risk. Risk of Ether loss after destroying the contract is also a concern for the developers. **2 respondents** are worried that people may send Ethers to the self-destruct contract, and these Ethers will be locked forever.

According to our survey feedback, we find that the *Selfdestruct* function might be risky for both contract developers and contract users. For contract developers, they need to pay more effort to ensure only specific people have permission to call the function, which might increase the cost of development. Besides, how to ensure no one will send Ethers to the self-destructed contract is also a big question. For contract users, their balance might be stolen by the contract owners, as the owners have the ability to transfer all the Ethers.

4 RQ2: REASONS FOR SELF-DESTRUCT

4.1 Motivation

In RQ1, we found that 66.67% of the developers will deploy a new version of the contract after destructing the old contracts. Therefore, by comparing the difference between the two versions of the contract, we can find the reasons why contracts destructed. Consider the following scenario.

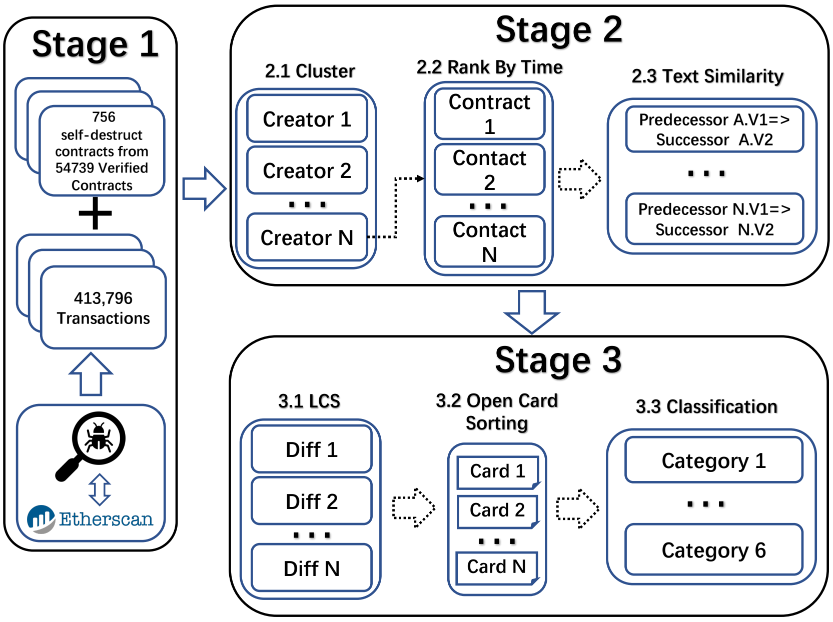


Fig. 1. Overview architecture of finding the self-destruct reasons

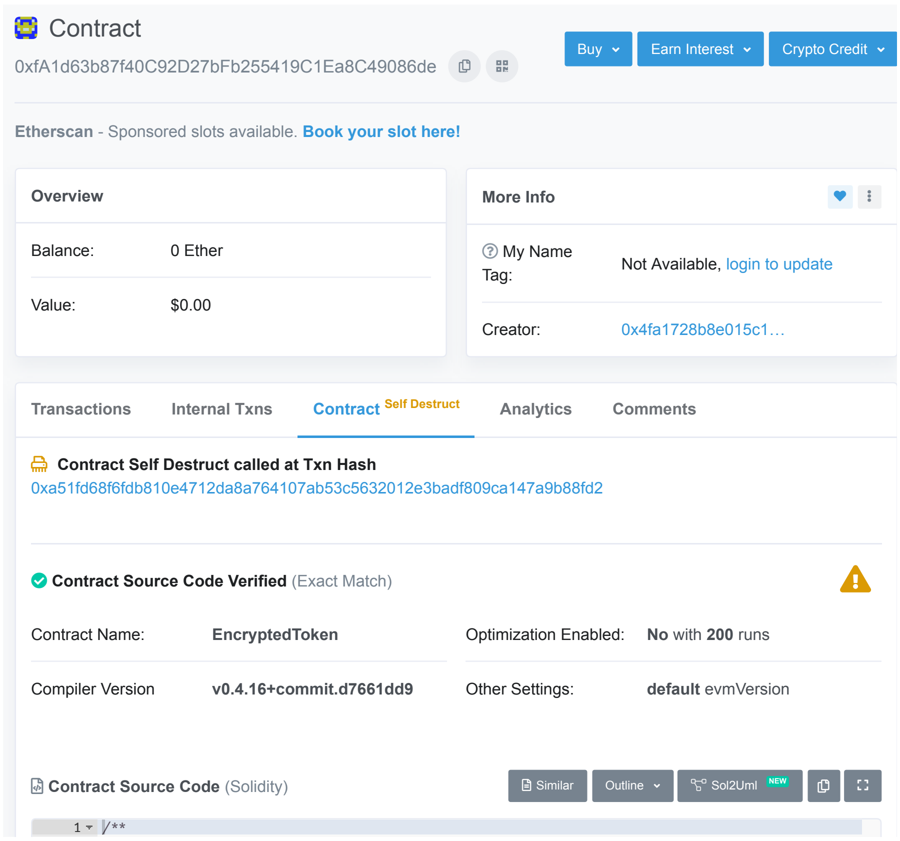
Bob is a smart contract developer. He developed a smart contract several weeks ago, and his company uses this contract to receive money from other companies. However, they find that a function in the smart contract does not limit the caller's permission, which can lead to serious security problems. Therefore, Bob has to destroy the contract by involving *Selfdestruct* function and deploy a new contract to the blockchain. The new contract adds a permission check to avoid this vulnerability. Bob and his colleagues try their best to inform other companies not to transfer Ethers to the self-destruct contract anymore. However, it requires a long time to inform all companies. Many users still transfer Ethers to the self-destruct contract, and all the Ethers send to the contract are lost forever. It causes a great financial loss to Bob's company.

From this scenario, we see that calling the *Selfdestruct* function may lead to great financial loss. Therefore, we should try to make contracts robust. If we tell Bob that many previous contracts are destructed because a function in the smart contract does not limit the callerfis permission, he might check whether his contract contains the same problem and can avoid this problem.

In this section, we compare the self-destruct contracts and their successor contracts to summarize reasons why contracts destructed. The reasons we identify can guide smart contract developers and help them refine their contracts.

4.2 Approach

Figure 1 depicts the detailed steps to identify the reasons why some smart contracts have been destructed. Our method consists of three stages. In the first stage, we crawl all verified contracts and their transactions from Etherscan. We crawled 54, 739 smart contracts altogether. We found that 2,786 (5.1%) of these smart contracts among 54,739 contracts contain a *Selfdestruct* function, and 756 (27.14%) contracts have been destructed. In the second stage, we first divide crawled contracts into several groups by their creators' addresses. In this case, we can find smart contracts that are



Contract
0xfA1d63b87f40C92D27bFb255419C1Ea8C49086de

Etherscan - Sponsored slots available. Book your slot here!

Overview

Balance:	0 Ether
Value:	$0.00

More Info

My Name Tag:	Not Available, login to update
Creator:	0x4fa1728b8e015c1…

Transactions | Internal Txns | Contract Self Destruct | Analytics | Comments

Contract Self Destruct called at Txn Hash
0xa51fd68f6fdb810e4712da8a764107ab53c5632012e3badf809ca147a9b88fd2

Contract Source Code Verified (Exact Match)

Contract Name:	EncryptedToken	Optimization Enabled:	No with 200 runs
Compiler Version	v0.4.16+commit.d7661dd9	Other Settings:	default evmVersion

Contract Source Code (Solidity)

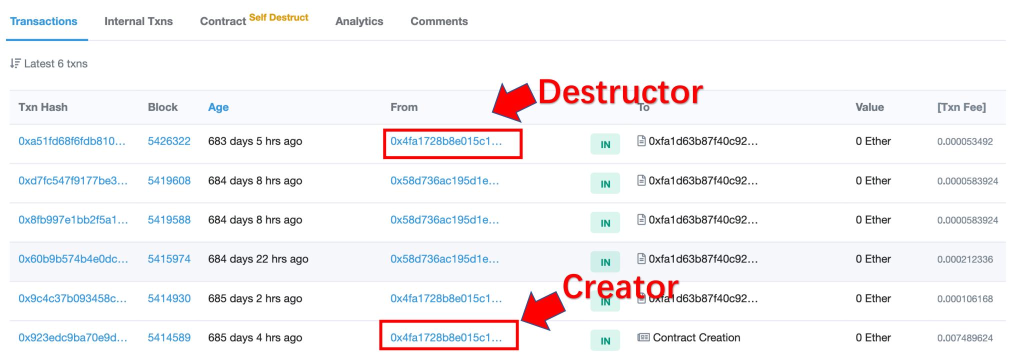
```
/**
```

Fig. 2. A smart contract on Etherscan

created by the same authors. We only use groups that contain self-destructed contracts and rank all the contracts in the same group by contracts' creation time. Then, we compute the code similarity of contracts in each group to find self-destruct contracts (also called predecessor contracts) and their successor contracts. In the last stage, we compare the difference of predecessor contracts and their successor contracts by using *open card sorting*. Finally, we summarized 6 reasons why contracts had been destructed.

4.2.1 Stage 1. Data Collection: Stage 1 is used to collect data for the following two stages. Our data contains three parts, i.e., *verified contracts*, *self-destruct contracts* ,and *contract transactions*.

Verified Contracts: Verified contracts are crawled from Etherscan. To crawl the source code of verified smart contracts from Etherscan, we first need to know the contract addresses of verified contracts. Figure 2 is a smart contract on Etherscan. By obtaining the contract address, we can easily download the source code, transactions ,and other information of the contract. The contract address list is provided by Etherscan (https://etherscan.io/contractsVerified). However, Etherscan only shows the last 500 verified contract addresses since Jan. 2019. The data used in this paper are crawled before Jan. 2019. We finally obtained 54, 739 verified contract addresses by the time of writing. After obtaining the contract addresses, we can crawl the source code from Etherscan directly.



Txn Hash	Block	Age	From		To	Value	[Txn Fee]
0xa51fd68f6fdb810...	5426322	683 days 5 hrs ago	0x4fa1728b8e015c1...	IN	0xfa1d63b87f40c92...	0 Ether	0.000053492
0xd7fc547f9177be3...	5419608	684 days 8 hrs ago	0x58d736ac195d1e...	IN	0xfa1d63b87f40c92...	0 Ether	0.0000583924
0x8fb997e1bb2f5a1...	5419588	684 days 8 hrs ago	0x58d736ac195d1e...	IN	0xfa1d63b87f40c92...	0 Ether	0.0000583924
0x60b9b574b4e0dc...	5415974	684 days 22 hrs ago	0x58d736ac195d1e...	IN	0xfa1d63b87f40c92...	0 Ether	0.000212336
0x9c4c37b093458c...	5414930	685 days 2 hrs ago	0x4fa1728b8e015c1...	IN	0xfa1d63b87f40c92...	0 Ether	0.000106168
0x923edc9ba70e9d...	5414589	685 days 4 hrs ago	0x4fa1728b8e015c1...	IN	Contract Creation	0 Ether	0.007489624

Fig. 3. Transactions of a Self-destruct Contract

Self-destruct Contracts: Finding whether a verified smart contract has been self-destructed is straightforward. If a contract has self-destructed, there will be a label (self destruct) given on Etherscan (see Figure 2). We found 756 self-destruct contracts from 54,739 verified smart contracts.

Transactions: Transactions on Ethereum record the information of the external world interacting with the Ethereum network. All the transactions can be found on Etherscan. We collect all 413,796 transactions of 54,739 verified smart contracts. Figure 3 is the transactions of a self-destruct contract. In the first transaction, we can find who deployed the contract (creator) and we can find who destructed the contract (destructor) in the last transaction.

4.2.2 *Stage 2. Pairs Generation:* The aim of stage 2 is to find *pair<predecessor contract, successor contract>*.

Step 2.1 Cluster: We first find the creator addresses of all the 54,739 verified smart contracts through their transaction. In this step, we inspect the first transaction of each smart contract as the first transaction contains the creator address and creation time. Then, we classify the contracts into several groups according to their creator addresses. If two contracts have the same creator address, they will be classified into the same group. We only choose groups that contain self-destruct contracts.

Step 2.2 Rank by Time: In this step, we first rank contracts in each group by their creation time, which can be obtained from the first transaction. Then, we can obtain several pairs; each pair is consisted of a self-destruct contract and a live contract. For example, one group contains five contracts, they are contract *a,b,c,d,e* and these five contracts are ranked by creation time. Contract b and d are the self-destruct contact in these five contracts. Finally, we output four pairs, i.e., *(b,c)*, *(b,d)*, *(b,e)* and *(d,e)*.

Step 2.3 Text Similarity: We compute the code similarity between two contracts to identify whether the later created contract is the successor contract of the self-destruct contract. SmartEmbed [33, 34] is the only tool that uses code embedding to find clone-related bugs in smart contracts, see Section 7). Although the tool is aimed at finding bugs, their first step computes code similarity between the given smart contract and history contracts. We modified the source code of SmartEmbed to compute the similarity between two contracts. If their similarity is larger than 0.6, they might be relevant and we assume the later created contract is the successor of the self-destruct contract. We also called this self-destruct contract as the predecessor contract of the successor contract. We found 436 self-destruct contracts have their successor contracts with 1513 <*predecessor contract, successor contract*> pairs. We note that 0.6 is a conservative threshold (original paper

Table 1. Reasons of Self-destruct.

Category	Description
Functionality Changes	Functionality changes for upgrading contracts.
Confusing Contract	Removing unused subcontracts, which might confuse users.
Limits of Permission	Adding permission checks for the contracts.
Unsafe Contracts	Removing the security problems of the contracts.
Unmatched ERC20 Token	Modifying the contract to make it suitable for the ERC 20 standard
Setting Changes	Changing the setting of the contract, such as token name, amount of ERC 20.

assumes similarity 0.95 are cloned); we might include many irrelevant pairs in our dataset, but it will not influence our result as we conduct a manual analysis in the subsequent step. Increasing the threshold can remove some irrelevant pairs to reduce the manual effort, but it might make us miss some true matching pairs.

4.2.3 Stage 3. Reason Generation: In this stage, we aim to find the reasons why contracts destructed.

Step 3.1 Longest Common Substring: Longest Common Substring (LCS) [30] algorithm is to find the longest string (or strings) that is a substring (or are substrings) of two or more strings. To reduce the manual efforts, we use LCS to find the different parts of the two contracts.

Step 3.2 Open Card Sorting: We follow the open card sorting [44] approach to analyze the smart contracts and summarize the reasons why they destructed. We create one card for each *pair<predecessor contract, successor contract>*. The detailed steps are:

Iteration 1: We randomly chose 20% of the cards, and two developers with 3 years of smart contract development discussed the reason why contracts destructed. If the reason of self-destruct is unclear, they omitted them from our card list. All the reasons are generated during the sorting.

Iteration 2: The same two smart contract developers independently categorized the remaining 80% cards into the initial classification scheme. We used Cohenfis Kappa [26] to measure the agreement between the two developers. Their overall Kappa value is 0.84, indicating strong agreement.

Step 3.3 Reason Generation: We finally categorized 6 reasons why contracts destructed. The information is shown in Table 1 and the detailed information is shown in the following subsection.

4.3 Reasons for Self-destruct

In this subsection, we give detail explanations of the 6 self-destruct reasons and their distribution in our dataset.

4.3.1 Definitions: **(1) Functionality Changes**: Functionality changes is the most common reason why smart contracts destructed. When new requirements appear or some requirements changed, some developers choose to deploy a new smart contract and the old version of the contract will be destroyed.

(2) Confusing Contract: Etherscan verification system we introduced in Section 2.7 can lead to problems. Contracts can add some subcontracts which are never called or inherited. These kinds of subcontracts will be removed when they are compiled to the bytecode. However, unused subcontracts can cause misunderstandings to users. In Ethereum, a contract address can only have one main contract. All the other subcontracts and libraries should be called or inherited by the main contract directly or indirectly. Otherwise, they are regarded as a confusing contract.

Example: There are four subcontracts in listing 3; the main contract is *Crowdsale*. *Pausable* is inherited by contract *Crowdsale* directly. *TokenCreator* is called by *Pausable*; so contract *TokenCreator* can be called by contract *Crowdsale* indirectly. No subcontracts have a relationship with *Token*. Therefore, *Token* is redundant as it will never be executed. However, Etherscan verified contract system cannot recognize the redundant code and will not remove the code of *Token*, which may cause misunderstandings to users that the contract can receive Ethers; as only *Token* contains payable fallback function[7] in these four subcontracts.

```
contract Crowdsale is Pausable{...}
contract Pausable{
    TokenCreator creator;...}
contract TokenCreator{...}
contract Token {
    function() payable{...}}
```

Listing 3. Example: Confusing Contract

(3) Limits of Permission: The predecessor contracts do not check permissions of the callers in some functions, which can lead to security problems. In the successor contract, they add modifiers to limit the permission.

Example: The predecessor contract in listing 4 does not limit the permission (L2); therefore, anyone can call the *Selfdestructs* function to kill the contract. In the successor contract, the function adds a modifier *onlyOwner* (L5) to check the permission. Only the owner of the contract can call this function.

```
1 //Predecessor Contract:
2 function Selfdestructs() payable public {
3     selfdestruct(owner);}
4 //Successor Contract:
5 function Selfdestructs() onlyOwner payable public {
6     selfdestruct(owner);}
```

Listing 4. Example: Limits of permission

(4) Unsafe Contracts: Previous works [22, 37, 40, 42, 45] introduced several security problems of smart contracts. For example, *Oyente* introduces four security issues, *Zeus* describes four security problems of smart contracts (see Section 7). We find many predecessor contracts contain security problems like *reentrancy*, which can lead to Ether loss. Developers usually fix these security issues in the successor contracts.

(5) Unmatched ERC20 token: ERC20 [3] is the most popular standard interface for tokens in Ethereum. If the implementation of token contracts does not follow the ERC20 standard strictly, the transfer between tokens may lead to errors. We find many predecessor contracts are token contracts but do not strictly follow the ERC20 standard, while their successor contracts follow the standard.

Example: ERC20 requires a transfer function to return a boolean value to identify whether the transfer is successful. However, the predecessor contract in listing 5 does not return anything. Users usually use third-party tools to manipulate their tokens and these tools capture token transfer behaviors by monitoring standard ERC20 method [24]. If the contract does not match the ERC20 standard, the token may fail to be transferred by third-party tools.

[7] A contract can receive Ethers only if it contains a payable fallback function.

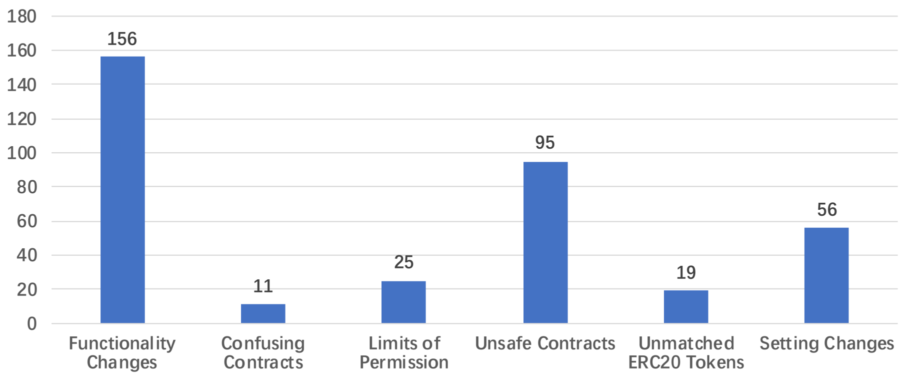


Fig. 4. Distribution of the reasons of self-destruct

```
//Predecessor Contract:
function transfer(address _to, uint _value);
//Successor Contract:
function transfer(address _to, uint _value) returns (bool success);
```

Listing 5. Example: Unmatched ERC20 token

(6) Setting Changes: Many contracts have some default values, especially for token related contracts. More specifically, token contracts need to set the total token supply, number of decimals the token use, and so on. If the developers want to change the total token supply, they have to kill the contract and deploy a new one.

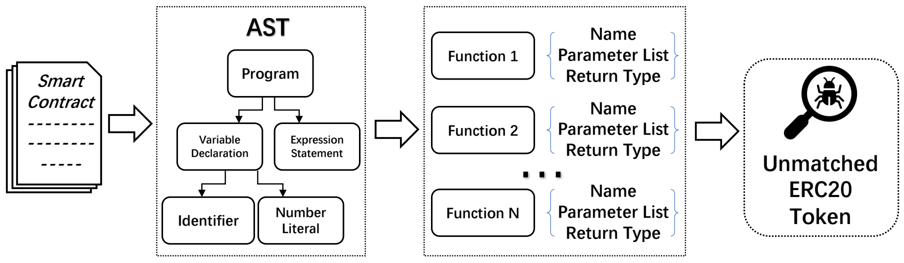


Fig. 5. Approach to Detect *Unmatched ERC20 Token*

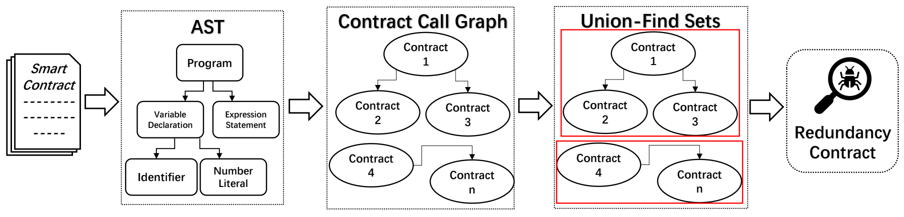


Fig. 6. Approach to Detect *Confusing Contract*

4.3.2 Distribution. We use SmartEmbed to find the pair<*predecessor contract, successor contract*>, and set the threshold value to 0.6, which might obtain some irrelevant pairs. After manually removing irrelevant pairs, we found 351 contracts (**some contracts have multiple self-destruct reasons**) for which we can identify the reason(s) why they are self-destructed, and give the distribution of the six self-destruct reasons in Figure 4.

It is clear that *Functionality Changes, Unsafe Contract*, and *Setting Changes* are the top three most popular reasons that lead to contracts destructed; the number are 156, 95 and 56, respectively. The number of other three reasons are similar, *Confusing Contracts* is the least popular reason that leads to the destruct of contracts. However, we find that only 12 contracts are *Confusing Contracts*, and 11 of them removed the redundant contracts in their successor version, which shows this problem is also important.

It should be noted that it is not easy to find all the security issues in our dataset. One the one hand, we only checked the security issues reported by *Oyente, Zeus, Mythril, Security, Maian* [5, 37, 40, 42, 45]. On the other hand, manually checking for security issues is very error-prone and time-consuming. To reduce the errors, we utilized tools by *Oyente, Zeus, Mythril, Security, Maian* and manually checked each contract it found. We first use the tools to check smart contracts. Then, two developers with 3 years of smart contract development experience manually identified whether the results are correct. If the reported results were different, they discussed to obtain the final result.

5 RQ3: LIFESCOPE: A SELF-DESTRUCT ISSUES DETECTION TOOL FOR SMART CONTRACTS

5.1 Motivation

In the previous section, we introduced six smart contract self-destruct reasons by comparing the difference between predecessor contracts and their successor contracts. Among these six reasons, *Functionality Changes* and *Change Setting* depend subjectively on the contract owner's requirements. Specifically, different developers might make different decisions of whether a smart contract should be self-destructed according to their requirements, even if the smart contracts are the same. It is thus hard to say these two reasons can affect the life cycle of smart contracts. However, smart contracts that contain the other four self-destruct reasons might have a short life span, as they can lead to unwanted behaviors of the smart contracts. Detecting whether a smart contract contains these self-destruct reasons might increase the life span of the contract. Manual analysis is time-consuming and error-prone. Therefore, designing a tool to detect whether a contract contains these self-destructed reasons is important. Security issues is a big concept. In the last section, we use the security vulnerabilities defined in previous works, e.g., *Oyente, Zeus, Mythril, Security, Maian* [5, 37, 40, 42, 45], to find unsafe contracts. These have already proposed several tools to detect security issues with high accuracy. The accuracy of *Zeus* is almost 100% according to their paper, and designing a more accurate and comprehensive security detecting tool is not the main target of this paper. In this case, we do not redevelop a tool to detect security issues introduced in these previous works.

5.2 Approach

We propose a tool named *LifeScope* to detect the remaining three issues, i.e., *Confusing Contract, Limits of Permission*, and *Unmatched ERC20 Standard* that can lead to contracts being destructed. Since the aim of *LifeScope* is extending the life span of a smart contract by finding the three self-destruct reasons, and smart contracts are immutable to be modified after deploying to the

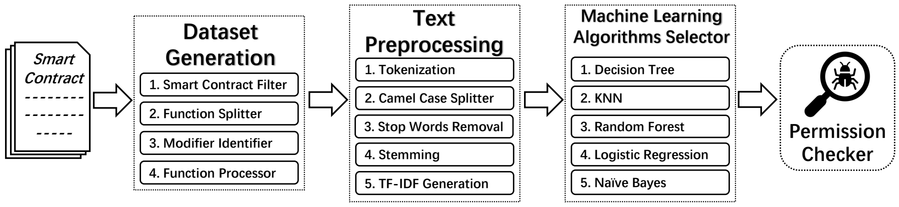


Fig. 7. Approach to Check Permissions

blockchain. Therefore, it is meaningless to detect the three self-destruct reasons through bytecode, although smart contracts are stored in the form of bytecode.

LifeScope detects the self-destruct issues at source code level, which utilizes AST (abstract syntax tree) to parse the smart contracts and extract related information to detect *Confusing Contract* and *Unmatched ERC20 Standard*. For *Limits of Permission*, *LifeScope* first transfers the contract to a TF-IDF representation and then utilizes machine learning algorithms to predict this problem. These three problems are not only limited to contracts that contain the *Selfdestruct* function. Any smart contracts can be analyzed with *LifeScope* to detect these three problems before deploying them to the Ethereum.

5.2.1 Unmatched ERC20 token. Figure 5 shows our approach to detect an *Unmatched ERC20 token*. We first extract the AST (Abstract Syntax Tree) of smart contracts by utilizing the *Solidity* compiler [13]. AST is a tree structure that contains the syntactic information of source code. By analyzing the AST, we can easily get the information of each function in the contract, including function name, parameter names, parameter types, and return types. Then, we compare whether related function matches ERC20 standard. For example, if a contract contains a function name *transfer*, then we check whether this function contains two parameters and their types are *address* and *uint256*. Besides, the function should have a return value, and the type of return value is *bool*.

5.2.2 Confusing Contract. Figure 6 is the approach to detect *Confusing Contract*. We first extract the AST of smart contracts. By parsing the AST, we can construct the call graph between contracts. In Ethereum, a contract address only has one main contract. All the other subcontracts, libraries, interfaces should be called or inherited by the main contract directly or indirectly. Otherwise, they are regarded as a confusing contract. In this case, if a contract is not a confusing contract, the call graph of the contract can not be spitted. We use the *Disjoint Set* algorithm [32] to get the number of parts of the call graph.

5.2.3 Limits of Permission. It is hard to prescribe functions need to check their permissions. Therefore, It is not easy to detect this issue by using programming analysis methods. We utilize a machine learning method to predict whether a function needs to check for its caller permission. Figure 7 describes the overall architecture that we used. The method contains three parts, i.e., *Dataset Generation*, *Text Preprocessing* and *Machine Learning Algorithm Selector*.

(a). Dataset Generation: The aim of this step is to extract pairs*¡func, permission¿* from smart contracts. In each pair, *func* is the source code of a function in the smart contract, and *permission* means whether the function needs to check the caller's permission. Since security vulnerabilities are ubiquitous in smart contracts on Ethereum [37, 40, 42**?**], some live contracts might also miss checking the permissions of the functions. This situation gets even worse in self-destruct contracts, as the reason for the destruct might be missing permissions. Therefore, it is not reliable to use these contracts as our ground truth. To ensure the correctness of our dataset, we should use contracts that correctly check their permissions for the contracts. However, it is not easy to ensure the correctness of the contract, and manually check whether a function needs to check its permission is also error-prone and subjective.

To obtain the dataset, we first rank all the alive verified contracts by their transaction numbers. Then, we choose all the contracts whose transaction numbers are larger than 500, as each transaction can be regarded as a test case for the contract. The more transactions, the less probability of having permission problems, as each transaction can be regarded as a test for the contract. After this step, 5,986 contracts remained, and we choose contracts whose balance have more than 1 Ether (The number of the contracts is 875). As the financial gain is the biggest motive behind most of the cyber attacks, a contract with high balance and transactions has lower probability of having permission problems.

After getting these contracts, we first remove the comments on the contracts and then split the contracts into functions. We obtain 29,313 functions from these 875 smart contracts. We then need to identify whether these functions contain modifiers, as the permission is usually checked by the modifier. If a function contains modifier, we remove the modifier from the function. For example, the original function is *function transferMoney(address addr) onlyOwner*{}. We remove the modifier *onlyOwner* from the function and only use *function transferMoney(address addr)*{} to training the machine learning model. In our dataset, we obtain 29,313 functions, with 4,393 of them needing to check permissions.

(b). Text Preprocessing: Before training the machine learning module, we need to transfer each processed function into a set of bag-of-words by the following steps: **(i) Tokenization**: Each processed function is divided into a list of words by punctuation and space that usually do not contain any information. For example *function transferMoney(address addr)*{} will be transfered into ”funciton”, ”transferMoney”, ”address” and ”addr” **(ii) Camel Case Spliter:** We separate function names, variable names and identifiers according the rules of Camel Case [16]. For example, ”transferMoney” will be separated into ”transfer” and ”Money”. Then, we transfer all the words to their lower case. **(iii) Stop Words Removal:** Stop words means meaningless words (e.g., ”to”, ”as”, ”is”). We adopt NLTK (a python library) stop words list in this step. We also remove tokens of less than 3 characters. **(iv) stemming:** this step is used to transfer words into their stem form. For example, ”running” is replaced by ”run”. In this paper, we use Porter’s stemmer [43] to transfer the words. **(v) TF-IDF generation:** We use TF-IDF (term frequency - inverse document frequency) [19] to represent each processed word in the function. This is described as:

$$w_{i,j} = tf_{i,j} * log(\frac{\#of functions}{df_i}) \tag{1}$$

Here, $w_{i,j}$ is the weight of the word i in the function j. $tf_{i,j}$ is the term frequency of word i in the function j. df_i is the number of functions that contain word i. Finally, each function is represented as $fun_j = (w_{1,j}, ...w_{i,j}, ..., w_{n,j})$.

(c). Machine Learning Algorithms Selection: Checking the permission of functions is a binary classification problem. We tried five popular machine learning algorithms to find an appropriate algorithm for predicting the permission. The five algorithms are *Decision Tree, KNN, Random Forest, Logistic Regression* and *Naive Bayes* . We found that the *Random Forest* algorithm obtains the best F-Score for this task. Therefore, we finally use *Random Forest* to predict whether a function needs to check for its callers permission.

Table 2. The accuracy of each machine learning algorithm.

Algorithm	Accuracy	Precision	Recall	F-Measure	AUC	TP	TN	FP	FN
Decision Tree	95.23%	84.91%	82.86%	83.85%	**0.9014**	363.9	2427.5	75.3	64.6
KNN	91.16%	79.68%	55.09%	65.11%	0.7631	242	2430.3	197.2	61.8
Random Forest	**96.10%**	**95.52%**	77.70%	**85.67%**	0.8853	341	2476.1	98.2	16
Logistic Regression	92.32%	85.73%	58.53%	69.55%	0.7841	257	2449.3	182.2	42.8
Naive Bayes	53.34%	23.32%	**92.35%**	37.22%	0.6941	405.6	1158	33.6	1334.1

5.3 Result

To evaluate LifeScope, we need to construct a ground truth dataset, which consists of two parts. The dataset of *Unmathched ERC20 Token* and *Confusing Contract* are generated during the process of finding self-destruct issues (see Section 4.2.3). The dataset of *Limits of Permission* is introduced in Section 5.2.3(a). To be noticed, the data shown in Figure 4 are 351 self-destruct smart contracts that can find their successor contracts, while we use all the self-destruct contracts as the ground truth of *Unmatched ERC20 token* and *Confusing Contract*.

(1). Unmatched ERC20 token: Our dataset has 756 self-destruct contracts. Among these contracts, 127 of them are discarded due to the unsupported compiler version. (LifeScope supports versions that higher or equal 0.4.25). So, there are finally 629 self-destruct contracts in our dataset. The two developers introduced before manually label the dataset and find 164 of them are ERC 20 token contracts. In these 164 ERC20 token contracts, 70 (11.13%) of them are *Unmatched ERC20 token*. *LifeScope* finds 70 *Unmatched ERC20 token*, with 0 false positive and negative.

(2). Confusing Contract: We use the same 629 self-destruct contracts as our dataset. The two developers manually label the dataset and find 12 (1.9%) of them are confusing contracts. *LifeScope* finds 12 confusing contracts, with 0 false positive and negative results.

(3). Limits of Permission: The dataset of *Limits of Permission* is introduced in section 5.2. We use *Precision*, *Recall* , *F-Measure* and *AUC* [35] to evaluate the performance of *LifeScope*, which are calculated as:

$$Accuracy = \frac{\#TP + \#TN}{\#TP + \#TN + \#FN + \#FP} * 100\% \tag{2}$$

$$Precision = \frac{\#TP}{\#TP + \#FP} * 100\% \tag{3}$$

$$Recall = \frac{\#TP}{\#TP + \#FN} * 100\% \tag{4}$$

$$F - Measure = \frac{2 * Precision * Recall}{Precision + Recall} * 100\% \tag{5}$$

TP (true positive) indicates the number which correctly predicts a function needs to add a permission check. TN (true negative) indicates the number which correctly predicts a function does not need to add a permission check. FP (false positive) and FN (false negative) indicate the number which incorrectly predict that a function needs or does not need to add a permission check. AUC (area under the curve) is calculated by plotting the ROC curve (receiver operator characteristic).

To better evaluate the results, we use a cross-validation method of training-testing sets. First, we divided our dataset into 10 parts of equal sizes. Then, we conduct the training using 9 parts of the dataset and 1 part for testing (2,913 cases for each part). We continue this process 10 times.

Finally, we use the average results to present the final results. Table 2 shows the results of five machine learning algorithms. *Random Forest* obtains the best result in predicting the permission of functions in the smart contracts, but *Decision Tree* gets the best result in *AUC*; the values are 0.9014. As a general rule, it is an acceptable performance for a classifier if the AUC score is larger than 0.7 [28, 29].

6 DISCUSSION

We first summarize the key implications of our work for researchers, practitioners, and educators. Then, we give 5 suggestions on how to better use *Selfdestruct* function according to the feedback of the survey. Finally, we summarize the main threats of validity.

6.1 Implications

6.1.1 For Researchers. ***Research Guidance.*** In this paper, we found 6 reasons why smart contracts destructed by comparing the difference between self-destruct contracts and their successor contracts. With the increasing number of smart contracts, researchers can apply our methods to find more problems that can affect the life span of smart contracts. Besides, Our study focuses on Ethereum smart contracts, but many other blockchain platforms also support the running of smart contracts, e.g., EOS [8]. These platforms have their own features; some of them also support actions similar to the *Selfdestruct* function. For example, developers can use a setcode action to remove smart contracts on *EOS*. The unique features of different blockchain platforms might lead to having different reasons for smart contract self-destruct. Researchers can use our methods to summarize the self-destruct reasons on these platforms.

Inconsistency. In this paper, we collected 54, 739 open-source smart contracts but only found 5.1% of them contain the *SelfDestruct* function. However, in our survey, 38.82% of the developers claim that they will add *SelfDestruct* functions to their contracts. There is an inconsistency between the practitioner's perception and their behavior (38.82% vs. 5.1%). The inconsistency might indicate that many developers admit the importance of *SelfDestruct* function but they give up on adding it during the actual development process. The reasons why developers do not add the *SelfDestruct* functions in actual developing process might have already been included in Section 3.3.2. Therefore, designing guidelines for using the *SelfDestruct* function might be helpful. Future work can aim to design guidelines, development models or tools to address these problems. We also give five suggestions in the Section 6.2.

In Section 4.3.2, we use 351 self-destruct smart contracts to give the distribution of the six self-destruct reasons. For 351 smart contracts we can find their successor contracts, which means developers destroyed the old contracts and deployed a new contract. There are 11 and 19 contracts in these 351 contracts that contain defects of *Confusing contracts* and *Unmatched ERC20 Tokens*, respectively. In Section 5.3, we found that the number of *Confusing contracts* and *Unmatched ERC20 Tokens* in all 657 self-destruct contracts are 12 and 70, respectively. The data shows that 11/12 (91.67%) of *Confusing contracts* are destroyed by the developers and replaced by deploying a new one. However, only 19/70 (27.14%) of *Unmatched ERC20 Tokens* contracts are destroyed and replaced with a new one. For this inconsistency, developers might assume *Confusing contracts* are more harmful than *Unmatched ERC20 Tokens*. Therefore, when developers find a *Confusing contracts* issue, they choose to destroy the contract and deploy a new one. For *Unmatched ERC20 Tokens*, developers might still choose to use the contract, and destroy the contracts once the life cycle is finished. However, we cannot find their successor contracts.

6.1.2 For Practitioners. Our work is the first that uses an online survey to collect feedback from smart contract developers on why they include or exclude *Selfdestruct* function. Their feedback shows that adding a *Selfdestruct* function can help developers transfer Ethers when emergency situations happen. However, using this function can also lead to several problems. To address the drawbacks of adding a *Selfdestruct* function, we give five suggestions in the next section. These can help developers better use the *Selfdestruct* function in their contracts. Smart contract developers can develop a smart contract according to our suggestions and open source the code for other developers to use. We also summarized 6 common reasons why contracts destructed and developed the LifeScope tool to detect 3 self-destruct reasons. Removing these problems might extend the life span of smart contracts.

6.1.3 For Educators. *Selfdestruct* is an important feature of smart contracts. However, most blockchain tutorials focus on teaching how to develop smart contracts and knowledge about blockchain [22]. Educators should pay more attention to these unique functions of smart contacts. For example, educators should mention the importance and drawbacks of adding a *Selfdestruct* function when they introduce this function. The feedbacks of our survey in RQ1 can provide good materials for them.

6.2 Towards More Secure Selfdestruct Functions

In Section 3, we summarized six reasons why smart contract developers exclude the *Selfdestruct* function from their contracts. In this part, we give five suggestions about how to better use *Selfdesturct* function according to the summarized worries.
Suggestion 1. Limit Usage Scenario: Adding *Selfdestruct* function can increase the complexity of the development and risk of attacks. Some smart contract developers claim that the *Selfdestruct* function is mainly used to remove the code and transfer Ethers. However, they do not need this function if there is no Ether in their contracts. Removing the contracts from the blockchain is also unattractive to them. Even if their contracts are attacked and controlled by attackers, they can discard the old contracts and deploy a new version. To reduce the risk and the workload of development, a *Selfdestruct* function is better to add in contracts that contain Ethers.
Suggestion 2. Permission Check: Calling a *Selfdestruct* function can lead to irreversible consequences. The contract has to check the permission of the caller in each transaction. A common method to check the permission is recording the owner's address in the constructor function. Then, checking whether the caller is the owner in each transaction.
Suggestion 3. Distribute the Rights and Modularization: The trust concern is an important reason why developers exclude a *Selfdestruct* function. The trust concern contains two parts according to the feedback, i.e., human related and code related concern.

For human related concern, users might worry that the owner of the contract can destruct the contract and transfer all the balance if the contract has a *Selfdestruct* function. For example, the *gambling* contract shown in Listing 1 claims that users can transfer 1 Ether to the contract. When the contract receives 10 Ethers, the contract will choose one user as the winner and transfer 9 Ethers to the winner as a bonus. However, if the contract has a *Selfdestruct* function, users might worry that the owner might transfer the money out at any time.

To reduce this kind of concern, we suggest the owner should distribute the rights of calling the *Selfdestruct* function. For example, all the users who transfer the Ethers to the contract should have the right to vote whether the contract should be killed. The voting steps can follow some consensus protocols, e.g., PoS [18], DPos [17]. Using PoS as an example, the more Ethers people contribute to the contracts, the more votes they have. This can increase the cost of attackers who want to stop the execution of a *Selfdestruct* function.

However, distributing the rights of calling a *Selfdestruct* function can increase code complexity, which is also a big concern according to our developer survey feedback. With the increase of code complexity, the probability of containing security vulnerabilities is also increasing, which leads to code related trust concerns for developers. To address these two concerns, we suggest that smart contract developers can open source and modularize this part of their code (Rights Distribution) to a library. Other developers can then help polish the code together, and can make the code easier to use in the future.

Suggestion 4. Delay Self-destruct Action: The Ethers sent to a self-destruct contract will be locked forever, which increases the risk of using *Selfdestruct* function. As we described in Section 4.1, the contract owner might find it difficult to inform all the users in a short time after the contract self-destruct. In this case, some users might send Ethers to the self-destruct contract and this may lead to financial loss. To address this problem, we suggest that the contract can delay the self-destruct action and throw an event to inform the users that the contract will self-destruct in the near future. On the one hand, delaying self-destruct action can give time for voting (Suggestion 3). On the other hand, it can provide time to inform users that the contract will be destructed.

```
bool isStopped = true;
modifier onlyOwner{ if(msg.sender != owner_addr) _;}
modifier stopContract{ if(isStopped == true) _; }
function changeState onlyOwner() {isStopped = !isStop;}
function moneyRelated() stopContract payable{ ... }
```

Listing 6. Example: Pause Functionality

Suggestion 5. Pause Functionality: The option in Suggestion 3 and 4 require time to implement. However, when a contract is being attacked, any delay might lead to enormous financial loss. In this case, pausing the functionality when performing the methods of Suggestion 3 and 4 is important. Listing 6 is an example of pausing functionality. The variable *isStopped* changes its state in function *changeState*, and this function can only be executed by the owner of the contract. Each money related function in the contract is controlled by the modifier *stopContract.* If the owner changes the variable *isStopped* to *true*, each money related function in the contract cannot be executed anymore, unless the owner changes *isStop* back to *false.*

6.3 Threats to Validity

Internal Validity. In RQ1, we sent our survey to 996 developers and received 88 responses. The response rate is 8.84%. We used the feedback of these 88 responses to summarize key reasons why developers include or exclude Selfdestruct function in their contracts. Due to the limited number of feedback. There might however still be other reasons we did not uncover in our survey. We collected all contributors emails from the top 100 most popular smart contract related projects. We also tried to make our survey as simple as possible and give 2 respondents $50 Amazon gift card to increase the response rate. We finally obtained a 8.84% response rate, which is also acceptable [48].

In RQ2, we use SmartEmbed to compute the similarity between smart contracts. If the similarity of two contracts larger than 0.6, we think they are a predecessor contract and its successor contract. The similarity threshold can influence the manual effort we need to pay. If the similarity is too large, we might miss some predecessor contracts and their successor contract. Otherwise, if the similarity is too small, we need to pay more effort to distinguishing whether the two contracts are relevant or not. The similarity threshold used in the paper of SmartEmbed is 0.95, and it found few contracts are relevant if the similarity is lower than 0.7. To reduce the number of unidentified relevant contracts, we conservatively reduce the threshold to 0.6. We used *Open Card Sorting* to find 6 common self-destruct reasons. Due to the limitation of our understanding of the smart contracts, we might miss some self-destruct reasons. To reduce the threat of human factors, we followed the process of card sorting strictly, and the developers all have rich experience (>3 years) in smart contract related research. Researchers can also use the same method we proposed in RQ2 to find other self-destruct reasons in the future.

In RQ3, we use contracts whose number of transactions are larger than 500, and balance is larger than 1 Ether as the ground truth. We regard these contracts having a low probability of having permission problems. However, it is still possible we might find some contracts in this group that have permission problems, but we believe the number of these functions is small. Since the total number of the functions used to train the algorithm is 29,313, a few noisey data will not affect the reliability of the final result.

External Validity. The smart contracts used in this paper were up to Jan. 2019. *Solidity*, the most popular programming language for the smart contract, is fast-growing. From Jan. 2019 to the time of writing, there are 11 versions updated and released. Many new features have been removed and added in these versions. Ethereum also might be updated in the future through a hard fork [15]. In this case, the self-destruct reasons might be changed because of a major update to Ethereum and Solidity. For example, Etherscan might upgrade its verification system. In this case, *confusing contracts* might never appear. Addressing this threat needs more research effort, but the method we proposed to find the self-destruct reasons is still working.

7 RELATED WORK

SmartEmbed [33, 34] is the first tool that uses a clone detection method to detect bugs in smart contracts. The tool contains a training phase and a prediction phase. In the training phase, their dataset contains two parts, i.e., source code database and bug database. The source code database consists of the source code of all the open source smart contracts in the Ethereum. The bug database records the bugs of each smart contract in a source code database. SmartEmbed first converts each smart contract to an AST(abstract syntax tree). After normalizing the parameters and irrelevant information on the AST, SmartEmbed transfers the tree structure to a sequence representation. Then, they use *Fasttext* [21] to transfer code to embedding matrices. Finally, they compute the similarity between the given smart contracts with contracts in their database to find the clone contracts and clone related bugs. Although the tool is aimed at finding bugs, their first step is computing the code similarity between the given smart contract and history contracts in their database. Therefore, we can modify their code to compute the similarity between two given smart contracts (used in RQ2).

Bartoletti et al. [20] found that the infamous Ponzi schemes migrated to the digital world. Many frauds use Ethereum to design Ponzi schemes contracts for earning money. They manually analyzed 1382 verified smart contracts on Etherscan and find 137 of them are Ponzi scheme contracts. Then, they divided these Ponzi scheme contracts into four categories, i.e., array-based pyramid schemes, tree-based pyramid schemes, handover schemes, and waterfall schemes. Bartoletti et al. opened their dataset to the public but do not provide a tool to detect whether a contract is a Ponzi scheme contract. To address this limitation, Chen et al. [47] proposed a method that uses a machine learning algorithm (XGBoost [23]) to distinguish Ponzi scheme contracts. They use account features and code features to train the module. The account features are extracted from the transactions, e.g., the number of payment transactions, the balance in the contracts. The code features can be obtained from contract bytecode. They count the frequency of each opcode in the contract bytecode. Both account features and code features do not need the source code of contracts. Therefore, their method can predict arbitrary contracts on the Ethereum.

Oyente [40] is the first tool for security examination for smart contracts based on symbolic execution. Their work introduces four security issues on smart contracts, i.e., mishandled exception, transaction-ordering dependence, timestamps dependence, and re-entrancy attack. To detect these security issues, Oyente first constructs a CFG (control flow graph) based on symbolic execution. After that, they design different rules to detect these four security issues. Kalra et al. [37] proposed a tool named Zeus, which can detect seven kinds of security problems; four of them are the same with Oyente; the other three issues are *failed send, interger overflow/underflow* and *transaction state dependence*. Zeus can detect security issues at the source code level. They use LLVM bytecode to represent the Solidity source and detect related patterns through LLVM bytecode. ContractFuzzer [36] is the first fuzzer to detect seven security issues in smart contracts. Four security issues are the same as Oyente; the other three issues are *gasless send*, *dangerous delegatecall* and *freezing ether*. ContractFuzzer utilizes ABI (abstract binary interface) of smart contracts to generate fuzzing inputs and defines test oracles to detect security issues.

Chen et al. [22] define 20 smart contract defects on Ethereum. They first crawl 17,128 Stack Exchange posts and use key words to filter solidity related posts. After getting Solidity related posts, they use *Open Card Sorting* to find 20 contract defects and divide them into five categories, i.e., *security, availability, performance, maintainability*, and *reusability defects*. According to their paper, although previous works define several security defects, they did not consider the practitioners' perspective. Therefore, they design an online survey to collect feedback from developers. The feedback shows that all the defined contract defects are harmful to smart contracts. They assign five impact levels the defined 20 contract defects. Defects with impact level 1-3 can lead to unwanted behaviors of contract, e.g., crashing or a contract being attacked.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we conducted the first empirical study on the use of the *Selfdestruct* function on Ethereum. To understand the smart contract developers' perspective, in RQ1, we designed an online survey to collect reasons from developers why they include and exclude the *Selfdestruct* function in their contracts. We summarized 5 reasons for including and 6 reasons for excluding *Selfdestruct* function in their contracts, respectively. The feedback also shows that 66.67% of developers will deploy a new version of the contract after destructing the old contract. According to this information, we propose an approach that can find the upgrade version of the self-destruct contracts in RQ2. After that, we used the *open card sorting* method and summarized 6 reasons why contracts might destruct. Three of them – *Unmatched ERC20 Token, Confusing Contract, and Limits of Permission* – can affect the life span of smart contracts. To detect these problems, we developed a new tool named LifeScope in RQ3, which reports 0 false positive / negative in detecting *Unmatched ERC20 Token* and *Confusing Contract*, and achieves an F-measure and AUC of 85.67% and 0.9014 for detecting the *Limits of Permission* issue. Finally, to help developers use the *Selfdestruct* function better, we give 5 suggestions based on the feedback of our survey and our smart contract analysis.

In the future, we plan to summarize more self-destruct reasons of smart contracts in Ethereum, as more contracts will be deployed, and new features will be added in Solidity / Ethereum. We will update LifeScope to detect the additional problems that lead to the self-destruct of contracts.

REFERENCES

[1] Apr., 2018 Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists/
[2] Apr., 2019 Decentralized application. https://en.wikipedia.org/wiki/Decentralized_application
[3] Apr., 2018 ERC20. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
[4] Aug., 2019 Ethereum Smart Contract Security Best Practices. https://consensys.github.io/smart-contract-best-practices/
[5] Aug., 2019 Mythril: Security analysis tool for EVM bytecode. https://github.com/ConsenSys/mythril
[6] Aug., 2019 Ripple Coin. https://ripple.com/
[7] Feb., 2019 Cryptokitties. https://www.cryptokitties.co/
[8] Feb., 2019 EOS. https://eos.io/
[9] Jan., 2019 Blockchain. https://en.wikipedia.org/wiki/Blockchain

[10] Jan., 2019 Cryptocurrency. https://en.wikipedia.org/wiki/Cryptocurrency

[11] Jan., 2019 Ethereum.org.

[12] Mar., 2018 EtherScan. https://etherscan.io/

[13] Mar., 2018 The Solidity Contract-Oriented Programming Language. https://github.com/ethereum/solidity

[14] Mar., 2018 Solidity Document. http://solidity.readthedocs.io

[15] Oct., 2019 Blockchain Hard Fork. https://www.investopedia.com/terms/h/hard-fork.asp

[16] Sept., 2019 Camel Case. https://en.wikipedia.org/wiki/Camel_case/

[17] Delegated Proof of Stake. https://lisk.io/academy/blockchain-basics/how-does-blockchain-work/delegated-proof-of-stake

[18] Sept., 2019 PoS: Proof of Stake. https://en.wikipedia.org/wiki/Proof_of_stake

[19] Baeza-Yates, Ricardo, and Berthier Ribeiro-Neto. Modern information retrieval. Vol. 463. New York: ACM press, 1999.

[20] Bartoletti, Massimo, et al. "Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact." Future Generation Computer Systems 102 (2020): 259-277.

[21] Bojanowski, Piotr, et al. "Enriching word vectors with subword information." Transactions of the Association for Computational Linguistics 5 (2017): 135-146.

[22] Chen, J., Xia, X., Lo, D., Grundy, J., Luo, X., and Chen, T. (2020). Defining Smart Contract Defects on Ethereum. IEEE Transactions on Software Engineering.

[23] Chen, Tianqi, and Carlos Guestrin. "Xgboost: A scalable tree boosting system." Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. 2016.

[24] Chen, T., Zhang, Y., Li, Z., Luo, X., Wang, T., Cao, R., ... and Zhang, X. (2019, November). TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (pp. 1503-1520).

[25] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhange, "Understanding ethereum via graph analysis," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 1484–1492.

[26] Cohen, Jacob. "A coefficient of agreement for nominal scales." Educational and psychological measurement 20.1 (1960): 37-46.

[27] Efanov, Dmitry, and Pavel Roschin. "The all-pervasiveness of the blockchain technology." Procedia Computer Science 123 (2018): 116-121.

[28] Fan, Yuanrui, et al. "The Impact of Changes Mislabeled by SZZ on Just-in-Time Defect Prediction." IEEE Transactions on Software Engineering (2019).

[29] Fan, Y., Xia, X., Lo, D., and Hassan, A. E. (2018). Chaff from the wheat: characterizing and determining valid bug reports. IEEE transactions on software engineering.

[30] Flouri, T., Giaquinta, E., Kobert, K., and Ukkonen, E. (2015). Longest common substrings with k mismatches. Information Processing Letters, 115(6-8), 643-647.

[31] Fowler, M. (2018). Refactoring: improving the design of existing code. Addison-Wesley Professional.

[32] Gabow, H. N., and Tarjan, R. E. (1985). A linear-time algorithm for a special case of disjoint set union. Journal of computer and system sciences, 30(2), 209-221.

[33] Gao, Z., Jayasundara, V., Jiang, L., Xia, X., Lo, D., and Grundy, J. (2019). SmartEmbed: A Tool for Clone and Bug Detection in Smart Contracts through Structural Code Embedding. In 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 394-397). IEEE.

[34] Gao, Z., Jiang, L., Xia, X., Lo, D., and Grundy, J. (2020). Checking Smart Contracts with Structural Code Embedding. IEEE Transactions on Software Engineering.

[35] Huang, J., and Ling, C. X. (2005). Using AUC and accuracy in evaluating learning algorithms. IEEE Transactions on knowledge and Data Engineering, 17(3), 299-310.

[36] Jiang, Bo, Ye Liu, and W. K. Chan. "Contractfuzzer: Fuzzing smart contracts for vulnerability detection." Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 2018.

[37] Kalra, Sukrit, Seep Goel, Mohan Dhawan, and Subodh Sharma. "ZEUS: Analyzing Safety of Smart Contracts." In NDSS, pp. 1-12. 2018.

[38] Kitchenham, Barbara A., and Shari L. Pfleeger. "Personal opinion surveys." Guide to advanced empirical software engineering. Springer, London, 2008. 63-92.

[39] Li, Xiaoqi, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. "A survey on the security of blockchain systems." Future Generation Computer Systems (2017).

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

[41] Nakamoto, S. (2019). Bitcoin: A peer-to-peer electronic cash system. Manubot.

[42] Nikolić, Ivica, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. "Finding the greedy, prodigal, and suicidal contracts at scale." In Proceedings of the 34th Annual Computer Security Applications Conference, pp. 653-663.

2018.
[43] Porter, Martin F. ”An algorithm for suffix stripping.” Program 14, no. 3 (1980): 130-137.
[44] Spencer, Donna. Card sorting: Designing usable categories. Rosenfeld Media, 2009.
[45] Tsankov, Petar, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. ”Securify: Practical security analysis of smart contracts.” In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 67-82. 2018.
[46] Tyagi, Pradeep K. ”The effects of appeals, anonymity, and feedback on mail survey response patterns from salespeople.” Journal of the Academy of Marketing Science 17, no. 3 (1989): 235-241.
[47] Chen, Weili, Zibin Zheng, Jiahui Cui, Edith Ngai, Peilin Zheng, and Yuren Zhou. ”Detecting ponzi schemes on ethereum: Towards healthier blockchain technology.” In Proceedings of the 2018 World Wide Web Conference, pp. 1409-1418. 2018.
[48] Xia, Xin, Zhiyuan Wan, Pavneet Singh Kochhar, and David Lo. ”How practitioners perceive coding proficiency.” In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 924-935. IEEE, 2019.