Understanding Code Reuse in Smart Contracts

Xiangping Chen[1], Peiyong Liao[2], Yixin Zhang[2], Yuan Huang[3,*], Zibin Zheng[3]
[1]School of Communication and Design, Sun Yat-sen University, Guangzhou, China
[2]School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China
[3]School of Software Engineering, Sun Yat-sen University, Zhuhai, China
chenxp8@mail.sysu.edu.cn {liaopy6,zhangyx95}@mail2.sysu.edu.cn {huangyuan5,zhzibin}@mail.sysu.edu.cn

***Abstract*—Smart contracts are programs that automatically execute on the blockchain system such as Ethereum. Everybody can write and deploy smart contracts on Ethereum, which causes a large collection of similar contracts via code reuse. In practice, code reuse in smart contract may amplify severe threats like security attacks, resource waste, etc. In this paper, we conduct an empirical study of code reuse in smart contracts for understanding the code reuse practice in the smart contract ecosystem. We first collect 146,452 open-source smart contract projects from Ethereum and then perform a detailed analysis. We first study how often the smart contract projects reuse and then we identify the top reused smart contracts and analyze how the developers revise smart contracts during reuse. Our research suggests that the code reuse in smart contract is quite frequent because about 26% contract code blocks are reused and the average time of reuse is 14.6. And the top reused contracts are almost all related to ERC20 token, which reveals that the current smart contract ecosystem is relatively homogenous. At last, we summarize 7 common types of code revision in smart contracts.**

***Index Terms*—Smart Contracts, Code Reuse, Code Clone, Solidity**

I. Introduction

Blockchain is a tamper-resistant and traceable distributed ledger that use a chained data structure to store data of transactions and blocks [1]. Ethereum [2], one of the most popular blockchain systems today, its most important innovation is the first introduction of smart contracts. A smart contract [3], [4] is a program that permanently stores and automatically executes on blockchain system like Ethereum to enable programmable transactions. Everyone can develop and deploy their own smart contracts to Ethereum and many people choose to open source their contract projects to prove that their contracts do not have malicious behavior or potential security vulnerabilities for user trust. Until May 2020, more than hundreds of thousands of open-source smart contracts have been published on Etherscan[1], and we believe this greatly facilitates code reuse in smart contracts, resulting in a large collection of similar smart contracts on Ethereum.

As smart contracts are becoming more and more popular, there are many studies concerned about the code reuse in smart contracts [5]–[7]. These studies employ multiple techniques (i.e., 5-grams [5], code embedding [6], undirected graph [7], etc.) to detect the code clone of smart contracts, and most of these studies found that the smart contract on the Ethereum is highly homogeneous. However, previous studies have mainly exposed the homogenization phenomenon in smart contracts, how code clone in smart contracts come into being and what the characteristic of contract code clone is are not well studied. For example, we do not know whether developers prefer to reuse contract source code directly or to reuse code with custom modifications, what type of smart contract is most likely to be cloned. There is a lack of in-depth analysis for code reuse in smart contracts.

We focus on analyzing the code reuse of smart contracts in this paper. The smart contracts on Ethereum are primarily written in Solidity language. Solidity [8] is a contract-oriented, high-level programming language that is influenced by many traditional object-oriented languages such as C++, Java. As a result, Solidity has many similar features to these common programming languages. In particular, there are three kinds of contract-level code blocks in Solidity, defined by three different keywords: *contract*, *interface*, *library*. The *contract*, similar to the class in object-oriented languages, contains some state variables and functions for accessing and modifying the state variables, corresponding to the data members and methods in the class respectively. The *interface* is the abstract contract whose functions cannot be implemented, analogous to the interface in Java. And the *library* is the contract used to encapsulate common functionalities and generally doesn't have any state variables. Therefore, in Solidity, contract-level code blocks are the basic complete code reuse units. For ease of description, we define these contract-level code blocks as "subcontracts" to avoid confusion with the term "contracts".

By the granularity of subcontracts, we explore the code reuse in open-source smart contracts from the overall to the detail, and we propose the following three research questions:

- **RQ1: How often do smart contract projects reuse subcontracts?** In this research question, we firstly study the proportion of the reused subcontracts on Ethereum, and then try to study the reuse of subcontracts in the within-project and the across-project scenarios.
- **RQ2: What are the subcontracts that are reused most?** In RQ2, we try to find out the subcontracts reused most based on the code clone detection approach, and we also try to find out the subcontracts reused most by inheritance [9].
- **RQ3: How do developers revise subcontracts in code reuse?** In this research question, we want to investigate into whether the developers revise the subcontract when

*Corresponding author.
[1]https://etherscan.io

they reuse it. Furthermore, we also want to know how the developers revise the smart contract to meet reuse requirements.

To explore the three research questions, we conduct an empirical study on the dataset which contains 146,452 smart contract projects in total. To detect the code reuse in the smart contract, we firstly analyze the abstract syntax tree of the smart contract to obtain its syntactic tokens, and then the syntactic tokens of a smart contract forming a token sequence is used to calculate the code similarity between smart contracts. After applying the code reuse detection on the dataset, some interesting results are disclosed. We find that over 26% subcontracts are reused with at an average of 14.6 times, and 91.11% of all projects have more than one subcontracts reused by other projects in RQ1. Meanwhile, we find that the most reused subcontracts are related to the ERC20 token applications [10] in RQ2. We also find that the developers are not inclined to make mass modifications to the reused subcontracts in RQ3.

The remainder of this paper is structured as follows: Section II introduces the overall framework of our approach and the code reuse detection rules. Section III presents our findings of the three research questions. Section IV and V presents the related work and threats to validity, respectively. At last, Section VI concludes the paper.

II. Study Design

A. Dataset and Overall Framework

Fig. 1 summarizes the overall framework of our study, which consists of the four phases, i.e., dataset collection, parsing and tokenization, pre-processing and detailed analysis.

Dataset Collection: Ethereum is one of the most popular smart contract platforms in which millions of smart contract projects have been deployed at the time of writing. Some of the developers may choose to open source their smart contract projects by publishing the source code to Etherscan platform to prove the security of their projects and gain the users' trust.

In this study, we collect all the open-source projects from Etherscan before August 2019, with 146,452 in total. The average number of code lines in the smart contract projects we collected is 497.56 and about 91.6% of the projects have no more than 1000 lines of code, which means that the code scale of smart contract project is relatively smaller than those in other fields, such as mobile apps or java applications, etc.

Parsing and Tokenization: we employ a Solidity language parser, *solidity-parser-antlr*[2] to extract the syntax information from source code in our dataset. This tool can either parse a complete project with multiple subcontracts or a single subcontract into an abstract syntax tree (AST). We use it to parse each subcontract in a project to its AST representation. Each node in the AST outputting from *solidity-parser-antlr* has a type string to represent its type of syntax structure. We hash a node in the AST by hashing its syntax structure type, specifically by hashing its type string. Then we tokenize each

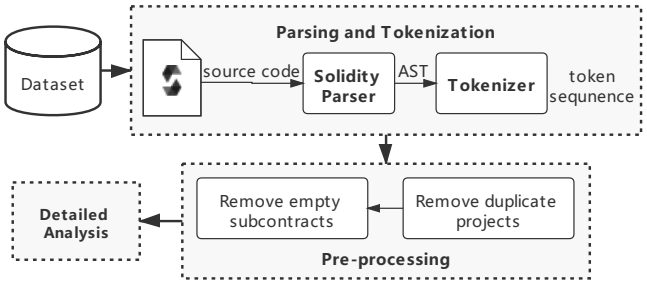

Fig. 1. The overall framework

subcontract by joining the hash values of all the nodes in its AST in pre-order traversal way to generate its token sequence, and the name of the subcontract is extracted from the root node of its AST. In others words, after tokenization, we obtained a mapping from the name of each subcontract to its token sequence for each project.

Pre-processing: the goal of pre-processing is to deduplicate the dataset according to a suitable criterion. We first define two projects to be identical if they have the same number of subcontracts as well as the corresponding subcontracts have the same name and token sequence. When we try to remove duplicate projects using this criterion, we find that about 85.4% of projects are duplicate, and these duplicate projects can be divided into 7,700 separate groups. Thus, for each group of duplicate projects, we further analyze how many identical projects the group contains and how many different creators created these projects. The creator information is also collected from Etherscan. And we conclude that there are two cases: (1) very few creators created a great number of identical projects, (2) a large number of creators created several identical projects separately. An example of the former is that a project[3] is repeated 42,500 times which are created by 2 creators. We suspect that the projects like these are published by automated deployment tools. For the latter case, most projects are developed for reusing token contract templates. Lots of creators issue their own tokens on Ethereum by copying these templates and then customizing the initialization parameters. Considering these two cases, we remove duplicate projects by their creators, i.e., we preserve one of the duplicate projects created by each creator rather than only one of each group of duplicate projects.

After dropping the duplicates, the number of projects in the dataset reduce from 146,452 to 52,951. There is a total of 286,976 subcontracts with an average of 5.42 per project. However, there are some empty subcontracts. An empty subcontract is a contract-level code block without any state variables, functions or other code lines inside it. The empty subcontracts are removed because they are meaningless for the code reuse analysis. The total number of non-empty subcontracts is 285,891, i.e., 235,435 contracts, 34,866 libraries and

[2]https://github.com/federicobond/solidity-parser-antlr

[3]https://etherscan.io/address/0x000067cecdaea7926d9ea6fe5bbb779ff35ac471#code

15,590 interfaces. Each project contains about 5.40 non-empty subcontracts on average.

Detailed Analysis: we perform the detailed analysis according to the three research questions proposed in the Introduction section. We first study the overall situation of code reuse in smart contracts in RQ1, and then we identify the most reused subcontracts in RQ2. Finally, in RQ3, we figure out the subcontracts with the most versions and analyze how developers revise these subcontracts during reuse.

B. Code Reuse Detection

In the experiment, we use the following three different detection rules to detect different kinds of code reuse.

- **Code Clone**: the names and the token sequences of two subcontracts are the same. The token sequence contains the syntax information of the subcontract, while the name usually contains semantic information about the functionality. We consider not only token sequences but also names because the subcontracts with the same name and token sequence are more likely to be reused based on copy-and-paste.
- **Code Reuse with Renaming**: two subcontracts have the same token sequence but different names. In some case, the developer may change the name of the reused subcontract due to the naming rules required by their institute or because there is a naming conflict if the name isn't changed. As a result, we compare the token sequence to detect possible code reuse with renaming.
- **Code Reuse with Revision**: only the names of two subcontracts are the same, and the syntax similarity between these two subcontracts are more than a specific threshold. The threshold we select in this paper is 0.9. And the one released earlier of the two is considered as the original version while the latter one is the corresponding revised version. The block number, a time-related attribute of Ethereum, is used to determine the order in which two subcontract versions are released and each deployed smart contract has a respective block number.

III. Result

A. RQ1: How often do smart contract projects reuse subcontracts?

Subcontract is the basic unit for a smart contract project, and this research question tries to gain a better understanding into the code reuse at subcontract level.

We want to understand the overall situation of code reuse by code clone in open-source smart contracts. We focus on the code reuse based on copy-and-paste and use the Code Clone detection rule to find out the code reuse instances of subcontracts. Since two subcontracts with the same name are not allowed in a smart contract project, we utilize the Code Reuse with Renaming rule to analyze the proportion of code reuse within project and across project.

Note that the developer may modify the source code while reusing it. The issue related to Code Reuse with Revision will be further discussed in RQ3.

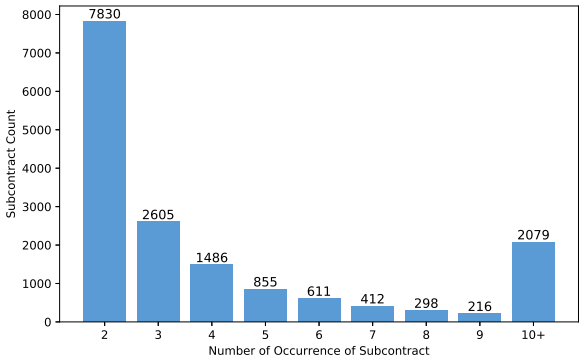

Fig. 2. Distribution of subcontract repetitions

(a) What is the proportion of subcontracts that are reused?

In our dataset, there are a total of 285,891 non-empty subcontracts in 52,951 smart contract projects. After applying Code Clone detection rule and filtering duplicated subcontracts in the dataset, we find 62,907 unique subcontracts. Each subcontract has about 4.54 duplications on average.

Fig. 2 shows the distribution of subcontract repetitions. Of the 62,907 subcontracts, 46,515 do not have any duplication, and 16,392 appeared twice or more. It indicates that about 26.06% subcontracts are reused about 14.60 times on average. The maximum times of reuse reaches 10,323. There are 2,079 subcontracts which are reused at least 10 times. These subcontracts must have some general purpose so that they can be reused in different projects.

(b) How are the subcontracts reused within and across projects?

In this question, we want to understand the reuse of subcontracts within the same project and across different projects.

In smart contracts implemented by Solidity, two subcontracts cannot be defined using the same name in a project. As a result, we use the Code Reuse with Renaming rule to detect the code reuse: only the token sequences of the subcontracts are required to be the same, while the names are not.

- within-project reuse

Subcontracts in a project that have the same token sequence are considered duplicate subcontracts. We find that the number of projects where within-project reuse exists is only 1,288. This number accounts for 2.43% of the total number of projects in our dataset. It shows that most subcontracts only appear once in a project, and code reuse within project is rare.

For a subcontract which is reused within project, it is reused 2.27 times on average. The maximum within-project reuse time is 14. Fig. 3 shows the distribution of the number of reuse time. We can see that about 80% (1,275 out of 1,567) of the subcontracts are reused exactly twice.

- across-project reuse

In smart contracts implemented using Solidity, the subcontracts are defined with different names in a project. As a result,

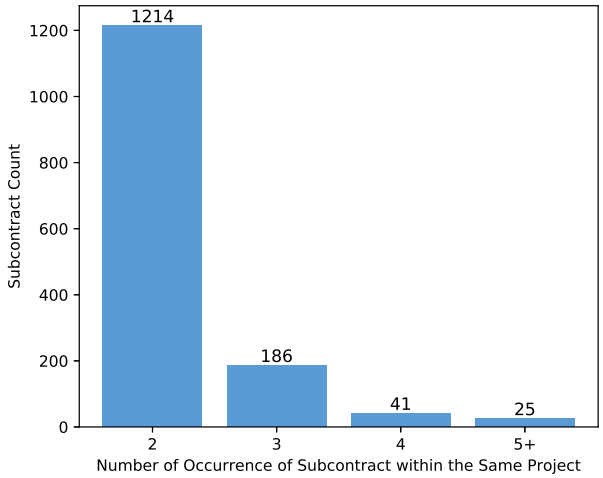

Fig. 3. Distribution of within-project subcontract repetitions

the reused subcontracts detected in question 1. (a) are reused across projects.

For comparison with within-project scenario, we also use the Code Reuse with Renaming rule to detect code reuse across projects. With the relax detection rule, more reused subcontracts are found. For each project, we traverse all the other projects and compare the subcontracts to determine whether the subcontracts are reused in other projects. We record the number of subcontracts that are reused across projects. An average of 4.6 subcontracts in each project are reused by other projects. Of all the projects in our dataset, 48,244 contain at least one subcontract which is reused by other projects, accounting for 91.11%. The ratio is much higher than that of the within-project reuse scenario.

B. RQ2: What are the subcontracts that are reused most?

In this research question, by counting the number of times each subcontract appears using the Code Clone detection rule, we find out the top 20 subcontracts with the most occurrences, i.e., (a) the top 20 most reused subcontract.

In addition, like object-oriented languages, Solidity supports inheritance for code reuse, which allows the developer to designate a new subcontract to extend an existing subcontract for inheriting the members of the existing one. The existing one is called base subcontract and the new one is called derived subcontract. Hence, we also identify (b) the top 20 base subcontracts that are inherited most to further investigate whether the inheritance is a primary way for reusing subcontracts.

(a) What are the top subcontracts that are reused by code clone?

Table I lists the top 20 most reused subcontracts and their reuse times. We can observe that some subcontracts are with the same name but different token sequences which are highlighted with the same color in Table I, such as *SafeMath*, *ERC20Basic*, *Ownable*, *StandardToken*.

TABLE I
THE TOP 20 MOST REUSED SUBCONTRACTS

	Subcontracts	**Reuse times**
1	ERC20	10323
2	SafeMath	6558
3	ERC20Basic	5761
4	Token	4999
5	Ownable	4773
6	ERC20Basic	4763
7	SafeMath	4605
8	StandardToken	4524
9	tokenRecipient	4501
10	SafeMath	4194
11	ApproveAndCallFallBack	2984
12	ERC20Interface	2968
13	SafeMath	2923
14	SafeMath	2750
15	StandardToken	2448
16	BasicToken	2429
17	StandardToken	2144
18	Ownable	1924
19	Ownable	1856
20	HumanStandardToken	1830

It is noted that these 20 subcontracts are widely used in smart contract development. For example, *SafeMath* is a library that provides safe arithmetic operations to avoid security vulnerabilities due to arithmetic overflows. *Ownable* is a subcontract for implementing ownership and access control. These subcontracts, *ERC20*, *ERC20Basic*, *ERC20Interface*, *Token*, *BasicToken*, *StandardToken*, are the common components in implementation of the ERC20 token standard by Ethereum community. Actually, *tokenRecipient* and *ApproveAndCallFallBack* are also related to the ERC20 token. *HumanStandardToken* comes from a token template. Therefore, we suggest that the current smart contract ecosystem is very monolithic and homogeneous, the major projects are oriented towards the ERC20 token.

In addition, we suspect that these subcontracts with most reuse times may come from open-source platforms such as GitHub, or from the smart contract communities. We try to use the source code of these most reused subcontracts to search the equivalent subcontracts from a repository, OpenZeppelin[4], which provides common smart contract components. OpenZeppelin is popular on GitHub and is frequently discussed in the smart contract communities. We find that 8 subcontracts in Table I (i.e., subcontract 1, 2, 3, 7, 10, 14, 15, 17) can be found in OpenZeppelin with the same source code.

(b) What are the top base subcontracts that are inherited?

In order to find out all the base subcontracts, we use the keyword *is*, which is used to establish subcontracts inheritance relationship in Solidity, to search in the AST of subcontracts.

[4]https://github.com/OpenZeppelin/openzeppelin-contracts

TABLE II
THE TOP 20 MOST REUSED BASE SUBCONTRACTS

	Base subcontracts	Reuse times
1	ERC20	9228
2	ERC20Basic	5463
3	ERC20Basic	4656
4	Ownable	4544
5	Token	3688
6	StandardToken	3275
7	ERC20Interface	2817
8	BasicToken	2424
9	StandardToken	2412
10	StandardToken	2028
11	SafeMath	1925
12	Ownable	1905
13	Ownable	1827
14	Owned	1474
15	Pausable	1446
16	owned	1313
17	BasicToken	1211
18	Owned	1209
19	Pausable	1130
20	Ownable	1074

TABLE III
THE TOP 20 SUBCONTRACTS WITH THE MOST VERSIONS

	Subcontracts	Number of versions	Percentage of modification
1	StandardToken	1010	2.429%
2	Crowdsale	756	1.887%
3	Token	665	2.584%
4	Ownable	574	0.096%
5	ERC20	536	2.873%
6	SafeMath	423	0.084%
7	TokenERC20	416	4.132%
8	BasicToken	409	0.374%
9	MintableToken	354	1.512%
10	Owned	326	0.213%
11	ERC20Token	264	1.574%
12	BurnableToken	179	0.315%
13	Pausable	167	0.103%
14	ERC20Interface	138	2.342%
15	ERC721	124	4.177%
16	owned	124	0.051%
17	MyAdvancedToken	119	1.024%
18	MyToken	103	10.256%
19	PausableToken	92	3.228%
20	token	92	1.563%

Then, we use the Code Clone detection rule to find out the most popular base subcontracts. The results are listed in Table II.

In Table II, we can see that all the most popular base subcontracts except *Owned*, *owned*, *Pausable* have already appeared in Table I. In other words, the most frequently reused subcontracts are those base subcontracts reused most by inheritance. This validates that inheritance is indeed the most common way of code reuse in Solidity.

Of these top 20 base subcontracts, 5 (i.e., subcontract 1, 2, 9, 10, 17) can be found in OpenZeppelin with the same source code. It indicates that the developers tend to take advantage of reliable and efficient subcontract implementations from well-known repositories by inheritance and then implement their own project-specific functionality based on the reused base subcontracts.

C. RQ3: How do developers revise subcontracts in code reuse?

For RQ3, we want to investigate into whether the developers revise the subcontracts they reused and how they revise them.

In the top 20 most reused subcontracts and the top 20 most popular base subcontracts in RQ2, we can see that some subcontracts such as *Ownable*, *ERC20Basic*, *StandardToken* appear more than once in each table. It indicates that more than one version of these subcontracts is widely reused. It is very common for the subcontracts to have the same name but different token sequences, since the developers may modify the subcontracts during reuse to get a version that fits their needs.

In this research question, we try to analyze how the subcontracts are reused with revision. We find out the top 20 subcontracts with the most versions, and then calculate the syntax similarities between different versions of each subcontract. The syntax similarity of two subcontracts is measured based on the longest matching subsequence of their token sequences:

$$similarity = \frac{length\,(longestSubSeq)}{max\_length\,(tokenSeq_1, tokenSeq_2)} \quad (1)$$

We employ the algorithm proposed in [11] to look for the longest matching subsequence. We firstly conduct the experiments considering whether developers revise the subcontracts during (a) reuse and during (b) reuse by inheritance, and finally analyze (c) how developers revise the base subcontracts.

(a) Do developers revise the subcontracts during reuse?

Table III shows that the top 20 subcontracts with the most versions are those that are widely reused, such as *Ownable*, *ERC20*, *ERC20Interface*, *SafeMath*, and various tokens like *StandardToken*, *BasicToken*, etc. These subcontracts have many different versions. We assume that these versions may appear when (1) the original publishers of subcontracts, e.g., OpenZeppelin, update the source code and create a new version; (2) some developers revise the subcontracts during reuse then publish the new version to Etherscan, and the revised versions are accepted by other developers.

In order to explore this question, for each version of each subcontract in Table III, we treat it as an original version and employ the Code Reuse with Revision rule to discover all

possible revised versions for it. Specifically, for each original version, we calculate the syntax similarities between it and other versions of the same subcontracts. If a version of the same subcontract has a similarity exceeding 0.9 to the original one and is published later, it is considered as a revised version modified by the developers from the corresponding original version.

After identifying the original and revised versions of each subcontract, we calculate the proportion of code modification for each subcontract using the formula 2:

$$PM_i = \frac{\sum_{j=1}^{Nv_i} Nr_{ij}}{\sum_{j=1}^{Nv_i} (f(O_{ij}) - 1) + \sum_{j=1}^{Nv_i} Nr_{ij}} \quad (2)$$

Nv_i, O_{ij} respectively denote the No. of versions and the j th version of the i th subcontract. Nr_{ij} denotes the No. of revised versions O_{ij} has. The function f maps an original version to its reuse times. And PM_i denotes the proportion of code modification of the i th subcontract. Thus, for the original version O_{ij}, there are $f(O_{ij}) - 1$ developers choose to reuse it without any revision, while Nr_{ij} developers revise it during reuse so that some new versions appear. For the i th subcontract, we separately accumulate the No. of developers reused without and with revision (i.e., $\sum_{j=1}^{Nv_i} (f(O_{ij}) - 1)$ and $\sum_{j=1}^{Nv_i} Nr_{ij}$), then compute a ratio to estimate the proportion of revision that occur. The results are shown in the fourth column of Table III.

The percentages of modifications of these 20 subcontracts are not remarkably high, which shows that the modifications to the code of smart contracts by developers during reuse are uncommon, compared with other traditional programming languages. One possible reason for this is the security requirements of smart contracts. The developers who do not have sufficient development experience may reuse without any modification. And on the other hand, smart contracts are mainly applied in token issuance which already has many mature templates without the need for developers to make any code revisions.

(b) Do developers revise the subcontracts that are inherited?

We preform similar experiments to investigate whether the developers revised the subcontracts that are inherited. We first find out the top 20 base subcontracts with the most versions, as shown in Table IV. Then we identify corresponding revised versions for each version of these base subcontracts, and calculate the proportions of the code modifications using (2).

It is noted that the first 17 subcontracts in Table IV are also listed in Table III. It indicates that there is a lot of overlap between the top 20 base subcontracts with the most versions (i.e., Table IV) and the top 20 subcontracts with the most versions (i.e., Table III), However, we notice that 2 of the other 3 subcontracts list in Table III whose names are prefixed with *My* (*MyAdvancedToken*, *MyToken*). For the subcontracts in Table IV, their names do not contain *My* because *My* is usually used as the prefix of a derived subcontract but not a base subcontract.

TABLE IV
THE TOP 20 BASE SUBCONTRACTS WITH THE MOST VERSIONS

	Subcontracts	**Number of versions**	**Percentage of modification**
1	StandardToken	917	2.358%
2	Ownable	556	0.100%
3	ERC20	401	2.733%
4	BasicToken	399	0.387%
5	Owned	313	0.221%
6	MintableToken	312	1.459%
7	TokenERC20	242	5.854%
8	Crowdsale	223	1.749%
9	Token	185	2.121%
10	ERC20Token	172	4.323%
11	BurnableToken	170	0.331%
12	Pausable	160	0.109%
13	SafeMath	132	0.180%
14	owned	114	0.057%
15	ERC721	105	4.730%
16	ERC20Interface	100	2.350%
17	PausableToken	89	3.443%
18	ERC223	71	7.360%
19	ERC20Basic	63	0.018%
20	Whitelist	49	0.820%

According to the programming naming conventions, a subcontract with the name *MyX* is generally a derived subcontract that inherits a subcontract with the name *X*. For example, the subcontract *MyToken* is likely to inherit the subcontract *Token*. Many developers have this naming convention, which would explain why the subcontracts like *MyToken* have more versions.

Another difference between these two tables is that the number of versions of each subcontract in Table IV is less than the number of versions of a subcontract with the same name in Table III. For example, the subcontract *SafeMath* in Table IV has 132 different versions, which is less than the subcontract *SafeMath* in Table III whose version number is 423. The reason for this is that a subcontract can be reused not only by inheritance. For example, the subcontract *SafeMath* is usually reused as *library* which cannot be used as base subcontract.

The proportions of code modifications are listed in the fourth column of Table IV. And we can reach the same conclusion that the developers are not inclined to make mass modifications to the reused subcontracts, not only because of the aforementioned reasons, but also because the subcontracts reused by inheritance do not generally involve functionality related to the developers' own projects so that the developers have no need to revise.

(c) How do developers revise the base subcontracts?

In this research question, we focus on analyzing how the developers revise the subcontracts that are reused most by

inheritance (i.e., are most inherited).

For each base subcontract in Table IV, we select its top 20 most reused versions as the original versions and then find out their revised versions. Specifically, a total of 355 subcontract version pairs are found. For each version pair, we manually compare their source code to analyze how the developers modify the subcontracts from the original one to its revision during reuse. The 355 samples of subcontract modification can be divided into 7 types, as list in Table V. For each revision type, we list the number of samples and use 1-3 reprehensive examples to illustrate the type of code change. The code change between versions are indicated in red.

Subcontract type: In Solidity, there are three types of subcontracts: *contract*, *interface*, and *library*. The developers may revise the subcontract types during reuse according to its application scenarios. For example, in case 1 of Table V, the developer changes the type of the subcontract *ERC20Interface* from *contract* to *interface*, because every function in *ERC20Interface* has only a definition without implementation. In addition, the subcontract type also conforms to the meaning of its name.

Inheritance: It is very common that a base subcontract also inherits other subcontracts. When the developers want to make the reused subcontract or its derivative subcontracts more functional or they have other implementations of the base subcontracts in their project, they may add or modify its base subcontracts during reuse, as shown in case 2 and 3 of Table V. A base subcontract *Ownable* is added to the subcontract *BurnableToken*. The base subcontract of *StandardToken* is changed from *Token* to *BasicToken* probably that *BasicToken* is very similar to *Token* with some project specific functionalities.

State variables: The state variables in a subcontract are similar to the data members of a class in an object-oriented programming language. The developers may remove or add state variables contained in the subcontracts during reuse. For example, in case 4 of Table V, the developer adds a new state variable, *initialSupply*, to represent the unnamed magic number, *130000000*, which is used directly in the reused subcontract. The modification can improve the readability and maintainability of the source code.

Functions: As in case 5 in Table V, the developers may also add or remove functions in the reused subcontract. A common case is adding a new function that is overloaded for an existing function in the reused subcontract.

Events: Events in Solidity are similar to the logging operation in other programming languages, which is a common programming practice to collect EVM runtime information. The developers may modify the events in the reused subcontracts to record more or less runtime information.

Visibility: Similar to object-oriented programming languages such as Java, the visibility of state variables or functions in smart contracts are specified using keywords: *public*, *private*, *external*, and *internal*. When reusing the source code, the developers may change these keywords when it is necessary to adjust the visibility of a function or variable.

Using library: The directive *using A for B* can be used to attach functions defined in subcontract *A* whose type must be library to any data type *B* in the context of a subcontract. For example, in case 10 of Table V, the developer has added such a statement to use the safe arithmetic operations provided by the library *SafeMath*, such as *safeAdd*, during reuse.

IV. Related Work

A. Empirical Studies on Code Reuse

The studies of code reuse go back a long way, and there are many studies of code reuse analysis for different programming languages, applications on different platforms, or software of different scales.

In general, the larger the software project, the more code reuse is needed to improve development efficiency. For example, to determine the extent of code reuse occurring in large open-source projects, Mockus et al. [12] collected the source code from several large open-source projects (mainly including several projects related to the popular distributions of Linux and BSD and several large individual projects), identified groups of files reused across these projects and determined the code most widely reused. Their research shows that over 50% of the files were used in more than one project and lots of files reused without any change, the most widely reused components were small, but some widely reused components involved hundreds of files.

Unlike the large open-source projects mentioned above, mobile applications are generally smaller in scale, but due to the popularity of mobile applications, studies on code reuse for mobile applications have become very interesting. Ruiz et al. [13] analyzed code reuse in the Android mobile app market along two dimensions: (a) reuse by inheritance, and (b) class reuse. Specifically, they studied on thousands of mobile apps across five different categories in the Android Market. And the results show that almost 23% of the classes inherit from a base class in the Android API and 27% of the classes inherit from a domain specific base class, on average 61% of all classes in each app category occur in two or more apps and 217 mobile apps are reused completely by another mobile app in the same category.

The Solidity language used in smart contracts has many similarities with object-oriented programming languages such as Java, so the study of code reuse for software projects developed in object-oriented programming languages is also informative for our study. Lewis et al. [14] designed an experiment to evaluate the impact of the object-oriented paradigm on software reuse. Their experiment concludes that (1) the object-oriented paradigm substantially improves productivity, (2) reuse without regard to language paradigm improves productivity, (3) language differences are far more important when programmers reuse than when they do not, and (4) the object-oriented paradigm has a particular affinity to reuse. To investigate whether open-source projects reuse third party code and how much white-box and black-box reuse occurs, Heinemann et al. [15] conducted an empirical study about software reuse in 20 open-source Java projects with a total

TABLE V
EXAMPLES OF CODE REVISION

ID	Code snippets before revision	Code snippets after revision	Type	Amount

1	contract ERC20Interface {
	...
	}
		interface ERC20Interface {
		...
		}
			Subcontract type	41

2	contract BurnableToken {
	function burn(uint256 v) public {
	...
	}
	}
		contract BurnableToken is Ownable {
		function burn(uint256 v) public onlyOwner {
		...
		}
		}
			Inheritance	141

3	contract StandardToken is Token {
	...
	}
		contract StandardToken is BasicToken {
		...
		}		

4	contract TokenERC20 {
	
	function TokenERC20() public {
	totalSupply = 130000000*10**uint256(decimals);
	}
	}
		contract TokenERC20 {
		uint256 initialSupply = 130000000;
		function TokenERC20() public {
		totalSupply = initialSupply*10**uint256(decimals);
		}
		}
			State variables	90

5	interface ERC721 {
	
	...
	}
		interface ERC721 {
		function transfer(address to, uint256 id) external;
		...
		}
			Functions	46

6	contract ERC20 {
	
	...
	}
		contract ERC20 {
		event Burn(address from, uint256 value);
		...
		}
			Events	76

7	uint256 totalSupply_;	uint256 public totalSupply_;	Visibility	147
8	function transferFrom(address, address, uint256) returns (bool);	function transferFrom(address, address, uint256) public returns (bool);		

9	contract SafeMath {
	function Add(...) public pure ...{
	...
	}
	}
		contract SafeMath {
		function Add(...) internal pure ...{
		...
		}
		}		

10	contract StandardToken is Token {
	
	function transfer ...{
	...
	balances[to] += value;
	...
	}
	}
		contract StandardToken is Token {
		using SafeMath for uint256;
		function transfer ...{
		...
		balances[to] = balances[to].safeAdd(value);
		...
		}
		}
			Using library	23

* *The red font indicates the code differences.*

of 3.3 MLOC (Million Lines of Code). They utilized static dependency analysis for quantifying black-box reuse and code clone detection for detecting white-box reuse. And the results indicate that software reuse is common among open-source Java projects and black-box reuse is the predominant form of reuse.

B. Code clone in Smart Contract

Smart contracts have gained a lot of attention in recent years and many researchers have studied the current status and explored the possible future development of smart contracts from various perspectives such as security [16]–[19], privacy [20], software engineering [21]–[23] and applications [24]–[26]. The work most related to this paper is about code clones in smart contracts.

In [5], Kiffer et al. obtained the bytecode of all contracts they collected from Ethereum to look for similarity and they found that less than 10% of user-created contracts are unique, less than 1% contract-created contracts are so, and more similarity can be found by clustering the contracts base on code similarity. Their results reveal that there is substantial code re-use in Ethereum and the bugs in reused contracts could have wide-spread impact.

Liu et al. conducted an attempt to clone detection of Ethereum smart contracts in [27]. They first introduced the concept of smart contract birthmark, a semantic-preserving and computable representation for smart contract bytecode, to reduce the clone detection to a computation of similarity between two contract birthmarks. And they implemented a tool called EClone to detect clone and evaluated it on Ethereum.

To help Solidity developers to find repetitive code and clone-related bugs in smart contracts, Gao et al. proposed a tool called SmartEmbed base on code embeddings and similarity checking techniques in [28]. This tool can efficiently identify code clones and clone-related bugs by comparing the similarities among the code embedding vectors for existing code in Ethereum and known bugs. They also applied SmartEmbed to

more than 22k contracts, and the results suggest that the clone ratio of Solidity code is close to 90% and 194 clone-related bugs can be found efficiently and accurately with a precision of 96%.

In [29], Liu et al. developed EClone, a semantic clone detector for Ethereum base on Symbolic Transaction Sketch, a set of critical semantic properties generated from symbolic transaction. They normalized the sketches of two smart contracts into numeric vectors with a same length for further similarity computation. Finally, they evaluated EClone to deployed Ethereum smart contracts and achieved an accuracy of 93.27%.

Different from these studies, our main goal is not to develop an efficient and accurate code clone detector, but rather to analyze code reuse in open-source smart contracts base on a code clone approach for a better understanding to Ethereum ecosystem.

V. Threats to Validity

In this section we focus on the threats that could affect the results of our case studies. The main threat to validity is the scale of the dataset. Since we need to analyze the reuse of the source code of smart contract, it requires that all the collected smart contracts should be open source. Then, we have collected 146,452 smart contract projects from Etherscan. However, it needs to note that these open-source smart contracts represent only a small part of the contracts on Ethereum considering a large number of non-open source contracts on Ethereum. In the future, we need to constantly crawl new open-source smart contracts from Etherscan to extend our repository.

Another threat to validity is the generalizability of our results. Because we focus on code reuse analysis of open-source smart contracts written by Solidity, our conclusions may not be applicable to smart contracts written in other languages, such as Serpent, LLL.

VI. Conclusion

For a better understanding of the smart contract ecosystem, we study the code reuse of open-source smart contracts on Ethereum. At the granularity of subcontracts, i.e., contract-level code blocks, we first analyze the overall profile of the code reuse in smart contracts, then identify the most reused subcontracts, and finally discuss if and how the developers revise the reused subcontracts while reusing them. Some interesting results are found, e.g., 26.06% of subcontracts in our dataset are reused at an average of 14.6 times, and 91.11% of all projects have more than one subcontracts reused by other projects. For the most reused subcontracts (including base subcontracts) we identify, almost all of them are related to Ethereum token-issuing applications. Lastly, we research the situation of revision to reused subcontracts by developers, the result is the revision behavior is infrequent, and we also conclude 7 ways to revise subcontracts are the modifications on subcontract type, inheritance, state variables, functions, events, visibility and library using. In the future, we will further crawl more open-source smart contracts to extend the dataset.

Acknowledgment

The research is supported by the Key-Area Research and Development Program of Guangdong Province (No. 2020B010164002), and the National Natural Science Foundation of China (No. 62032025, No. U1811462).

References

[1] S. Nakamoto and A. Bitcoin, "Bitcoin: A peer-to-peer electronic cash system," *URL: https://bitcoin.org/bitcoin.pdf*, 2008.

[2] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, 2014.

[3] N. Szabo, "The idea of smart contracts," *Nick Szabo's Papers and Concise Tutorials*, vol. 6, 1997.

[4] Z. Zheng, S. Xie, H. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, "An overview on smart contracts: Challenges, advances and platforms," *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020.

[5] L. Kiffer, D. Levin, and A. Mislove, "Analyzing ethereum's contract topology," in *Proceedings of the Internet Measurement Conference 2018*, 2018, pp. 494–499.

[6] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Transactions on Software Engineering*, 2020.

[7] N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, "Characterizing code clones in the ethereum smart contract ecosystem," in *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 654–675.

[8] R. M. Parizi, A. Dehghantanha *et al.*, "Smart contract programming languages on blockchains: an empirical evaluation of usability and security," in *International Conference on Blockchain*. Springer, 2018, pp. 75–91.

[9] E. Nasseri, S. Counsell, and M. J. Shepperd, "An empirical study of evolution of inheritance in java oss," 2006.

[10] S. Somin, G. Gordon, and Y. Altshuler, "Network analysis of erc20 tokens trading on ethereum blockchain," in *International Conference on Complex Systems*. Springer, 2018, pp. 439–450.

[11] R. Wettel and R. Marinescu, "Archeology of code duplication: Recovering duplication chains from small duplication fragments," in *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*. IEEE, 2005, pp. 8–pp.

[12] A. Mockus, "Large-scale code reuse in open source software," in *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*. IEEE, 2007, pp. 7–7.

[13] I. J. M. Ruiz, M. Nagappan, B. Adams, and A. E. Hassan, "Understanding reuse in the android market," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 2012, pp. 113–122.

[14] J. A. Lewis, S. M. Henry, D. G. Kafura, and R. S. Schulman, "An empirical study of the object-oriented paradigm and software reuse," in *Conference proceedings on Object-oriented programming systems, languages, and applications*, 1991, pp. 184–196.

[15] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck, "On the extent and nature of software reuse in open source java projects," in *International Conference on Software Reuse*. Springer, 2011, pp. 207–222.

[16] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[17] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, "Smartshield: Automatic smart contract protection made easy," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 23–34.

[18] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[19] W. Chen, X. Guo, Z. Chen, Z. Zheng, Y. Lu, and Y. Li, "Honeypot contract risk warning on ethereum smart contracts," in *2020 IEEE International Conference on Joint Cloud Computing*. IEEE, 2020, pp. 1–8.

[20] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 839–858.

[21] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 442–446.

[22] M. Knecht and B. Stiller, "Smartdemap: A smart contract deployment and management platform," in *IFIP International Conference on Autonomous Infrastructure, Management and Security*. Springer, 2017, pp. 159–164.

[23] Y. Huang, Q. Kong, N. Jia, X. Chen, and Z. Zheng, "Recommending differentiated code to support smart contract update," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 260–270.

[24] H. F. Atlam, A. Alenezi, R. J. Walters, G. B. Wills, and J. Daniel, "Developing an adaptive risk-based access control model for the internet of things," in *2017 IEEE international conference on internet of things (iThings) and IEEE green computing and communications (GreenCom) and IEEE cyber, physical and social computing (CPSCom) and ieee smart data (SmartData)*. IEEE, 2017, pp. 655–661.

[25] C. Dong, Y. Wang, A. Aldweesh, P. McCorry, and A. van Moorsel, "Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 211–227.

[26] G. G. Dagher, J. Mohler, M. Milojkovic, and P. B. Marella, "Ancile: Privacy-preserving framework for access control and interoperability of electronic health records using blockchain technology," *Sustainable cities and society*, vol. 39, pp. 283–297, 2018.

[27] H. Liu, Z. Yang, Y. Jiang, W. Zhao, and J. Sun, "Enabling clone detection for ethereum via smart contract birthmarks," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 105–115.

[28] Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo, and J. Grundy, "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)*. IEEE, 2019, pp. 394–397.

[29] H. Liu, Z. Yang, C. Liu, Y. Jiang, W. Zhao, and J. Sun, "Eclone: Detect semantic clones in ethereum via symbolic transaction sketch," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 900–903.