Analysis of Source Code Duplication in Ethereum Smart Contracts

Giuseppe Antonio Pierro
Centre Inria
Lille Nord Europe
Lille, France
giuseppe.pierro@inria.fr

Roberto Tonelli
Dep. of Mathematics and Computer Science
University Of Cagliari
Cagliari, Italy
roberto.tonelli@dsf.unica.it

***Abstract*—The practice of writing smart contracts for the Ethereum blockchain is quite recent and still in development. A blockchain developer should expect constant changes in the security software field, as new bugs and security risks are discovered, and new good practices are developed. Following the security practices accepted in the blockchain community is not enough to ensure the writing of secure smart contracts.**

The paper aims to study the practice of code cloning among the smart contracts by analyzing two corpora. The first corpus, the "Smart-Corpus", includes smart contracts already deployed in the Ethereum blockchain. The second corpus, the "Open-Zeppelin's Solidity Library", is supervised by a community of developers who constantly take care to increase the security and efficiency of the smart contracts included in the corpus. From the comparative analysis of the corpora, we observe that the smart contracts developers frequently duplicate the code by cloning already existing smart contracts which are not part of the "OpenZeppelin corpus". In particular, we found that 79.1% of smart contracts contain duplicated code and only 18.4% of smart contracts reuse the code by implementing a smart corpus belonging to the OpenZeppelin repository.

The paper discusses the advantages and the disadvantages of code duplication in the Ethereum blockchain ecosystem, and suggests to refer to the smart contracts of the OpenZeppelin's Solidity Library. The Ethereum blockchain community can indeed benefit from using the tested code presented in OpenZeppelin's Solidity Library to increase its security.

***Keywords*—code duplication; smart contract; Ethereum blockchain**

I. INTRODUCTION

Smart contracts are programmable contracts that can be automatically executed when certain pre-defined conditions are met. The smart contracts are applicable to a variety of fields, such as IoT, digital identity, supply chain, healthcare, business process management, etc. [1].

Smart contracts also have important economic implications in the fields where they are or could be used. The cost of smart contracts' failure can be high, and remedial actions can be difficult. Many researchers therefore emphasized the importance of adopting good practices in writing smart contracts' source code [2], [3]. It is indeed not enough to defend the smart contracts against known vulnerabilities [4]. Instead, the blockchain developer needs to learn a new practice of smart contracts' development [5], [6], [7], [8].

A bad programming habit could be the "code cloning". We define "code cloning" in smart contracts as the act of duplicating identical or near identical pieces of already written source code. According to previous literature [9], some problems related to code cloning are:

- the code cloning's tendency to create inconsistencies in the process of update, which hinder maintenance and contribute to the aging of the software.
- the increasing size of the source code due to code cloning.

In the Ethereum blockchain, the cost of Gas to deploy the smart contract is also related to the size of the source code [3]. Usually, smart contracts with duplicate code can be refactored with a saving in terms of Gas [10], [11], but this means that further work for developers is required [12].

A motivation for this work is precisely the fact that code clones make the smart contract source files very hard to consistently modify. For instance, if a smart contract has several functions created by code duplication with a slight modification, the software developer needs to carefully modify all the other functions in the smart contracts when a fault is found in one function.

The paper addresses the following research questions:

- Q1: What is the percentage of duplicated code on smart contracts deployed in the Ethereum blockchain? Is it increasing or decreasing over the last 5 years?
- Q2: What might be some causes of smart contracts' source code cloning in the Ethereum blockchain?

The paper aims to answer the questions, analyzing two corpora of smart contracts and also discussing some cases of clones refactoring.

II. BACKGROUND

Manual source code copy and modification is often used by programmers as an easy means for the reuse of some functionality. Nevertheless, such a practice produces duplicated pieces of code or clones whose maintenance might be difficult. Duplicated codes are therefore good candidates for system redesign [13].

We can define two types of code clone:

- "Local code clone" indicates that the same piece of source code is present in different parts of the same smart contract.

- "Global code clone" indicates that the same piece of source code is present in different smart contracts deployed in the Ethereum blockchain.

Both types of code clones have been considered as a bad software development practice [14]. Some of the reasons are:

- they can potentially cause maintainability problems, for example when a cloned code fragment needs to be changed, it might be necessary to align such a change across all clones.
- Code duplication increases the size of the code, extending compile time, expanding the size of the executable and thus increasing the costs in the Ethereum Blockchain.
- Code duplication often indicates design problems, such as missing inheritance or procedural abstraction which hampers the addition of functionalities.

Previous research pointed out that source code clones are introduced for reasons such as:

- making a copy of a code fragment is simpler and faster than writing the code from scratch. [15],
- writing a code with time pressure leads to plenty of opportunities for code duplication, especially in industrial software development contexts, [16], [9].

We proposed other hypotheses in the case of smart contracts, as the Ethereum blockchain has other specificities with respect to other programming ecosystems. Some smart contract developers may copy the code from the code of a smart contract that has already proved to be successful, deeming to be successful in their turn. Another reason may be due to the fact that the Ethereum blockchain does not have an official package manager to deploy smart contracts. A package manager is a programming language tool to create project environments which allow to easily import external dependencies [17].

The reasons why code clones appear in source code have been analyzed in other programming languages [18] and code clones' detection tools have been proposed as well [15], [19]. Methods for clone resolution include refactoring [20] and meta-level techniques [21].

The aim of our research is to investigate the use of source code clone information as a basis for smart contracts source code refactoring. Indeed, sometimes removing clones could be so difficult, that it would be better to maintain the duplication, but sometimes clones could also be good candidates to redesign the system, as they represent duplicated code whose consistent maintenance might be difficult to achieve [22]. They also form possible explicit connections among components that share the same piece of code and functionalities. Detection of source code clones in large software systems, such as JDK, FreeBSD, NetBSD, Linux, and many other systems has been investigated in the past by [23] while clone elimination or reduction in programming languages, such as Java, has been investigated by Balazinska [13].

III. Related Work

M. Kondo et al. [24] studied the phenomenon of smart contracts cloning in Ethereum Blockchain. They found that 79.2% of the smart contact studied are clones and that the percentage of clones among newly created smart contracts continues to increase over time. Moreover, they identified 26.3% of all 165,005 code blocks extracted from their corpus as identical to OpenZeppelin code blocks. Most of these code blocks belong to the ERC20 OpenZeppelin category. Our study confirmed their findings, by investigating another corpus of smart contracts source code. Also, their analysis regarded smart contracts deployed on the Ethereum blockchain until February 2018. We extended their research to the months until December 2020.

N. He et al. [25] proposed a classification of the code clones among the smart contracts deployed in the Ethereum blockchain. They analyzed a corpus of 10 million smart contracts, deployed from July 2015 to December 2018. Interestingly, they found that a large number of duplicated contracts suffered from the vulnerability issues inherited from the original contracts. Some of their results are confirmed by our research. We extended the analysis to smart contracts deployed in the last two years and, in addition, we also considered the code duplication inside the same smart contract.

M. Araoz et al. [26] present OpenZeppelin, one of the most popular packages to develop secure smart contracts. OpenZeppelin contains a collection of code blocks (subcontracts, libraries, and interfaces) that can be used as building blocks to develop blockchain-based applications. For instance, it includes implementations of the ERC20 standard, mathematical libraries (e.g., SafeMath), contract lifecycle management contracts (e.g., Pausable contract), and even cryptography utilities. As of December 28, 2020, the project has 2,218 commits, 243 contributors, and 8.9K stars in its GitHub repository. The development team at OpenZeppelin aims to produce high-quality code to be reused by smart contract developers. The team adheres to the following development principles: in-depth security, simple and modular code, clarity-driven naming conventions, comprehensive unit testing, pre-and-post-condition sanity checks, code consistency, and regular audits. Ultimately, code blocks from OpenZeppelin can be interpreted as "certified" pieces of code that are developed by a community that strives for security and performance. In particular, these code blocks are meant to be reused without modification. Their work is very relevant for us, because a solution to code duplication can come from the libraries proposed by their repository. In our work, we precisely show how code cloning can be avoided by simply using their libraries.

IV. Research Methodology

A. Research questions

The research has been lead by the following questions:

- Q1: What is the percentage of duplicated code on smart contracts deployed in the Ethereum Blockchain? Is it increasing or decreasing over the last 5 years?
- Q2: What might be some causes of source code cloning in the Ethereum blockchain?

B. Data Collection

To collect smart contracts source code we used the "Smart Corpus" [27]. "Smart Corpus" is a repository made of 30K smart contract data (source codes, ABIs and byte codes). Unlike the existing repositories which make available the source code in a laborious way, "Smart Corpus" instead makes this task easier and faster. Indeed, one of the main advantages of using Smart Corpus lies in the fact that it can reduce the costs in performing the smart-contract static analysis. For our analysis we have not considered all the smart contracts in the corpus but only a part. This choice was made for two reasons: 1) to have a homogeneous distribution of smart contracts with respect to the programming language version (the pragma) and with respect to the year in which the smart contracts were installed on the Ethreum blockchain. 2) to reduce the amount of time needed to compare all smart contracts searching for code clones.

C. Data Cleaning

Before performing the algorithm to identify smart contracts code clones, we cleaned the data collected in the "Smart Corpus" based on some considerations of Ethereum blockchain's specific features. The users have no permission to change the smart contracts deployed in the Ethereum blockchain. Indeed, if the user wants to correct a bug in a smart contract, s/he is forced to redeploy and correct the same smart contract by using a new unique address. As a result, on the Ethereum blockchain there might be two or more almost identical smart contracts with different addresses. The fact that different addresses refer to the same smart contract let us suppose that many smart contracts might simply be "trials" or smart contracts deployed in the blockchain to test and eventually modified them on the basis of the test results.

Fortunately, the smart corpus used to analyse the source code [27] in addition to the smart contract's address, contains the smart contract creator's address. The smart contract creator's address is the address of the smart contract owner, who has deployed the smart contract on the Ethereum blockchain. This piece of information allows us to test the hypothesis that many smart contracts are deployed from the same smart contract creator address with few differences. Indeed, for the purpose of this analysis, we excluded similar smart contracts having the same smart contract creator's address: 28% of smart contracts, 2134 of 7623 were excluded for this reason.

D. Data Reporting

We distinguished two types of smart contract clones. The local smart contracts clones defined as source code duplication inside the same smart contract and the global smart contracts clones defined as code duplication among all smart contracts deployed in the Ethereum blockchain. We used a script based on simple string matching, to find code clones on the same smart contract. The script performs the following steps:

- the source code is slightly transformed using string manipulation operations that remove spaces, empty lines and comments;
- Then, all the lines of the source code are compared among them to find code clones.

We chose the source code line as the minimal unit on which to perform the algorithm. As an example, the line of a smart contract source code

```
if( a > b && a > c) { // if else statement
```

is condensed to

```
if(a>b&&a>c){
```

We used a different approach to find code clones among different smart contracts, as for instance many DL-based code clone detection methods [?]. L. Jiang et al. [28] developed an algorithm named Deckard. Deckard algorithm is based on Abstract Syntax Tree (AST) of the source code of a program to find exact or close matches of subtrees of another AST source code. The algorithm code is available at the following address: https://github.com/skyhover/Deckard.

As an example, consider the following two smart contracts fragments:

```
for (uint i=0; i<arrayLength; i++) {
  totalValue += mappedUsers[addressIndices[i]];
}

for (uint j=0; j<customersLen; j++) {
  total += customers[addressIndeces[j]];
}
```

The parse trees for the two code fragments are identical, because the code differs only in the identifier names and literal values.

We used Deckard algorithm for the following reasons:

- It is language-independent and works in any programming language that has a context-free grammar (CFG), such as Solidity, the programming language used to write smart contracts.
- It has already been used to analyze clones in smart contracts, and we could thus compare the results of our study with previous studies that used the same algorithm [24].

For the Deckard algorithm configuration we set the variable "min_tokens" equal to 50 and the variable "similarity" equal to 0.79. The variable "similarity" is the threshold for tree similarity. Tree similarity is determined as a function of tree editing distance, which is the minimal sequence of edit operations (either relabel a node, insert a node, or delete a node) required to transform one parse tree into another. Following previous literature recommendations, we set the variable value at 0.79 [28].

V. Results and Discussion

Tables I, II present the results of our analysis. The first table I presents the average percentage of both smart contracts' local and global code duplication, grouped per year in which the smart contracts were deployed in the Ethereum blockchain. The second table II presents the average percentage of both smart contracts' local and global duplication, grouped per pragma version of the smart contracts deployed in the Ethereum blockchain. Pragma is a directive that specifies

TABLE I
AVERAGE PERCENTAGE OF DUPLICATION DIVIDED PER YEAR

Year	2016	2017	2018	2019	2020
Local Clone (in the same smart contract)	68%	63%	61%	57%	51%
Global Clone (among different smart contracts)	37%	49%	68%	76%	82%
Number of contracts analysed	1213	1984	1567	1342	983

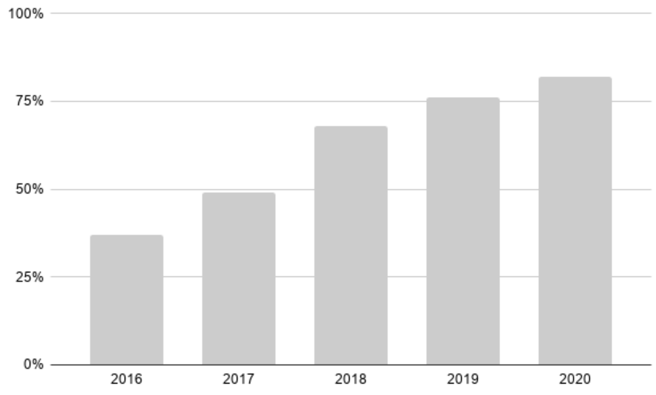

Fig. 1. Evolution of the percentage of global clones among the smart contracts for every year.

what compiler version to use to compile the source code to obtain the runtime byte-code. The results of the trends of both local and global code clones, presented in the two tables, do not change. This can be explained by the fact that there is a direct correlation between the date and the different versions of the Solidity language.

As to the first research question (Q1), Table I shows how the percentage of both local and global source code cloned is very significant, though the percentage of smart contracts cloned in the Ethereum blockchain is based on a limited sample. Moreover, the results are compatible with previous studies [24], even when analyzing a different time window by including the years 2019-20 which had not previously been studied. Table I also shows that the percentage of global clones increases over time and pragma versions.

As to the second question (Q2), we need to separately discuss the two cases, global and local code clones.

An increase of "global code cloning" over time can be

TABLE II
AVERAGE PERCENTAGE OF DUPLICATION DIVIDED PER PRAGMA VERSION

Pragma	0.3.x	0.4.x	0.5.x	0.6.x	0.7.x
Local Clone (in the same smart contract)	68%	64%	63%	61%	57%
Global Clone (among different smart contracts)	37%	49%	68%	76%	82%
Number of contracts analysed	767	1912	2043	898	657

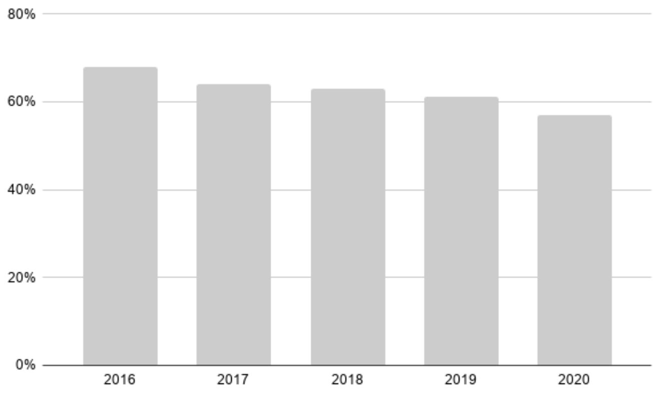

Fig. 2. Evolution of the percentage of local clones among the smart contracts for every year.

observed. We propose the following explanation for this phenomenon:

Firstly, from a manual inspection we discovered that some of these cloned contracts refers to highly-active smart contracts. In the context of this research, we considered as highly-active smart contracts the smart contracts having an overall number of transactions (both in input and in output) involving the contract greater than 100 per day. An example of highly-active smart contract is the CryptoKitties smart contract [29]. It is reasonable to assume that smart contract developers clone highly-active contracts with the hope to achieve the same commercial success [30]. In detail, the booming success of the popular Ethereum game Decentralized Application (DApp) "Crypto Kitties" [29] (a game in which players collect and breed digital cats) in the late 2017 led to the development of a plethora of smart contract clone versions. Some of these Ethereum game DApps are "Crypto Dogs" [24] and "Crypto Alpaca" [31]. However, neither of these smart contracts' clones ever achieved the same popularity of CryptoKitties. Secondly, differently from other more oldest programming languages in use today (C, C++, Java, PHP, Python, ECMAScript), Solidity does not have a package manager.

A package manager is a programming language tool to create project environments which allows software developers to import external dependencies. By using a package manager the developer does not need to reinvent the wheel or to copy and paste the code from other projects to implement new features. Software developers who use other programming languages, such as Java and ECMAScript, can employ this practice. For example, Java software developers use "Apache Maven"(https://maven.apache.org/), Script software developers use "node package manager" (https://www.npmjs.com/). Indeed, a package manager for Solidity would be very useful to add functionalities to smart contracts using code certified by the open source community. Moreover the package manager can perform a security review of the project's dependency tree. Audit reports contain information about security vulnerabilities in the dependencies and can help to fix a vulnerability by providing simple-to-run commands and recommendations for further troubleshooting. Recently, a non-official package manager to develop smart contracts in the Ethereum blockchain has been proposed. The project is currently under development and the proposal is available at the following address: https://docs.ethpm.com/

An decrease of “local code cloning” over time can be observed, as shown by Figure 2. To better understand this decreasing trend, we made a manual inspection of the cloning fragments. From a manual inspection we discovered that some local cloning is easily removable by following the recommendations given by the Solidity programming language documentation available at the following address: https://docs.soliditylang.org/.

Listing 1 shows an example of a local clone (see lines 17 and 22).

Listing 1. Smart contract with local clone (see lines 17 and 22)

```
1
2   pragma solidity >=0.7.0 <0.8.0;
3
4   contract BasicAccessControl1 {
5
6       address payable admin;
7
8       constructor() {
9           admin = msg.sender;
10      }
11
12      function publicFunction1() external {
13          // ...
14      }
15
16      function privateFunction1() external {
17          require(msg.sender == admin, 'Only Admin');
18          // ...
19      }
20
21      function privateFunction2() external {
22          require(msg.sender == admin, 'Only Admin');
23          // ...
24      }
25
26  }
```

According to the official Solidity language documentation, the code duplication can be avoided by using a “function modifier”. A function modifier is a Solidity construct which is used as a pattern to change the behavior of some functions, and in many cases, to restrict them. Listing 2 displays the improved version of Listing 1. In detail, it is possible to avoid the code repetition, by including the logic in modifiers (see lines 24-27 of Listing 2) and applying them to a function (see lines 16 and 20 of Listing 2).

Listing 2. Smart contract without local clone.

```
1
2   pragma solidity >=0.7.0 <0.8.0;
3
4   contract BasicAccessControl1 {
5
6     address public admin;
7
8     constructor() {
9         admin = msg.sender;
10    }
11
12    function publicFunction1() external {
13        // ...
14    }
15
16    function privateFunction1() external onlyAdmin() {
17        // ...
18    }
19
20    function privateFunction2() external onlyAdmin() {
21        // ...
22    }
23
24    modifier onlyAdmin() {
25        require(msg.sender == admin, 'Only Admin');
26        _;
27    }
28  }
```

We propose the following explanation for the decreasing trend of local code repetition over the years.

Firstly, it is reasonable to assume that over time there are more and more tools that help the smart contract developers to write code following the “coding best practices”. Coding best practices are a set of informal rules that the software development community employs to improve the quality of softwares [21]. Indeed, in recent years, several tools have been proposed and published in academic papers. Some of these tools are SmartCheck [32], SmartAnvil [33] and PASO [34]. These tools were not available in early versions of Solidity programming language. They share the ability to help the user to detect bad coding practices, such as code repetition and/or possible vulnerabilities.

Another plausible reason to explain this trend lies in the fact that, in general, source code reuse in object-oriented programming languages is made possible through different mechanisms, such as inheritance, shared libraries, object composition, and so on. In the first version of Solidity, some of these mechanisms were not provided to the smart contracts' developers. For example, the “Interface Contract” was introduced only starting from Solidity version “0.4.11”. “Interface Contracts”, similarly to the interfaces used in object-oriented languages, allow decoupling the definition of a contract from its implementation, providing better extensibility. Indeed, when a Contract Interface is defined, the implementations of a new Contract can be provided for any existing functions without modifying their declarations.

As a project grows, the need for additional functionality increases. Some of these functionalities (see lines 21–24 of Listing 2), that are cloned among different smart contracts (the global code clones), can be found in various libraries, such as OpenZeppelin. However, Solidity does not have a package manager. Instead the user who aims to reuse the code provided by OpenZeppelin, needs to look for the module in its code repository. This is not a major obstacle for programmers with long experience who know how to search the code in a repository such as OpenZeppelin, but it could be a problem for less experienced programmers, who may search in the web for already written code to implement additional functionalities in their smart contract without a package manager which helps to solve their problems. Some of the already written code may not be updated with the last version of Solidity and present some vulnerabilities.

Listing 3 displays the improved version of the listing 2 by removing the function modifier, which is one of the most copied code among the smart contracts deployed in the Ethereum blockchain, and by importing the "Ownable.sol" module from the OpenZeppelin project (see line 4 of Listing 3). The "Ownable.sol" module provided by the OpenZeppelin project makes the modifier "onlyOwner" available, which can be applied to private functions to restrict their use to the smart contract's owner.

Listing 3. Smart contract which imports the openzeppelin module.

```
1
2  pragma solidity >=0.7.0 <0.8.0;
3
4  import "@openzeppelin/contracts/ownership/Ownable
       .sol";
5
6  contract BasicAccessControl1 is Ownable{
7
8      address public admin;
9
10     constructor() Ownable{}
11
12     function publicFunction1() external {
13         // ...
14     }
15
16     function privateFunction1() external
           onlyOwner() {
17         // ...
18     }
19
20     function privateFunction2() external
           onlyOwner() {
21         // ...
22     }
23 }
```

VI. Conclusion

The paper showed that different types of code clones can be found in smart contracts. Out of 7500 smart contracts analysed, there are about 80% of smart contracts that contain code which is duplicated from other smart contracts deployed in the Ethereum blockchain. In the same smart contracts corpus, we found that the code clones within the same smart contract are about 40%. Based on previous literature, we know that maintaining these clones is an error-prone task and a potential threat to the system's overall security.

From the analysis done in the Smart Corpus [27] we have seen that the two kinds of "code clone" have opposite trends. While the local code repetition is decreasing over the years and it is inversely proportional to the pragma version number, the global code repetition is increasing over the years. Based on the data, we provided some explanations on the possible causes of code duplication for both types of code clones. The proliferation of clones may be caused by the desire to copy successful Ethereum DApps or by the lack of a package manager tool that allows smart contracts developers to easily import external dependencies without the need to copy and paste existing code. A qualitative study involving smart contracts developers (e.g., a survey or a series of interviews) could provide additional insights into the causes of code cloning in the Ethereum blockchain.

References

[1] S. Huh, S. Cho, and S. Kim. Managing iot devices using blockchain platform. In *2017 19th International Conference on Advanced Communication Technology (ICACT)*, pages 464–467, 2017.

[2] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F. Wang. Blockchain-enabled smart contracts: Architecture, applications, and future trends. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 49(11):2266–2277, 2019.

[3] Marco Ortu, Matteo Orrú, and Giuseppe Destefanis. On comparing software quality metrics of traditional vs blockchain-oriented software: An empirical study. In *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 32–37. IEEE, 2019.

[4] A. Mense and M. Flatscher. Security vulnerabilities in ethereum smart contracts. In *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications and Services*, iiWAS2018, page 375–380, New York, NY, USA, 2018. Association for Computing Machinery.

[5] L. Marchesi, M. Marchesi, and R. Tonelli. ABCDE - agile block chain dapp engineering. *CoRR*, abs/1912.09074, 2019.

[6] Felix Hartmann, Xiaofeng Wang, and Maria Ilaria Lunesu. Evaluation of initial cryptoasset offerings: the state of the practice. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 33–39. IEEE, 2018.

[7] Felix Hartmann, Gloria Grottolo, Xiaofeng Wang, and Maria Ilaria Lunesu. Alternative fundraising: success factors for blockchain-based vs. conventional crowdfunding. In *2019 IEEE international workshop on blockchain oriented software engineering (IWBOSE)*, pages 38–43. IEEE, 2019.

[8] Martina Matta, Ilaria Lunesu, and Michele Marchesi. Bitcoin spread prediction using social and web search media. In *UMAP workshops*, pages 1–10, 2015.

[9] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, Feb 2010.

[10] V. P. Ranganthan, R. Dantu, A. Paul, P. Mears, and K. Morozov. A decentralized marketplace application on the ethereum blockchain. In *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*, pages 90–97, 2018.

[11] G. A. Pierro, Henrique Rocha, Roberto Tonelli, and Stéphane Ducasse. Are the gas prices oracle reliable? a case study using the ethgasstation. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 1–8. IEEE, 2020.

[12] Alessandro Murgia, Roberto Tonelli, Michele Marchesi, Giulio Concas, Steve Counsell, Janet McFall, and Stephen Swift. Refactoring and its relationship with fan-in and fan-out: An empirical study. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 63–72. IEEE, 2012.

[13] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings Seventh Working Conference on Reverse Engineering*, pages 98–107, 2000.

[14] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 97–106, 2002.

[15] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[16] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, page 86, USA, 1995. IEEE Computer Society.

[17] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The spack package manager: bringing order to hpc software chaos. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[18] J. Svajlenko and C. K. Roy. Evaluating clone detection tools with bigclonebench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 131–140, 2015.

[19] M. Rieger and S. Ducasse. Visual detection of duplicated code. In *ECOOP Workshops*, 1998.

[20] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, page 50–59, New York, NY, USA, 1974. Association for Computing Machinery.

[21] B. H. Liskov. A design methodology for reliable software systems. In *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I*, AFIPS '72 (Fall, part I), page 191–199, New York, NY, USA, 1972. Association for Computing Machinery.

[22] Lodovica Marchesi, Michele Marchesi, Giuseppe Destefanis, Giulio Barabino, and Danilo Tigano. Design patterns for gas optimization in ethereum. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 9–15. IEEE, 2020.

[23] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[24] Masanari Kondo, Gustavo A. Oliva, Zhen Ming (Jack) Jiang, Ahmed E. Hassan, and Osamu Mizuno. Code cloning in smart contracts: a case study on verified contracts from the ethereum blockchain platform. *Empirical Software Engineering*, 25(6):4617–4675, Nov 2020.

[25] Ningyu He, Lei Wu, Haoyu Wang, Yao Guo, and Xuxian Jiang. Characterizing code clones in the ethereum smart contract ecosystem. *CoRR*, abs/1905.00272, 2019.

[26] M. Araoz et al. zeppelin_os: An open-source, decentralized platform of tools and services on top of the evm to develop and manage smart contract applications securely. 2017.

[27] M. Marchesi et al. An organized repository of ethereum smart contracts' source codes and metrics. *Future Internet*, 12(11):197, Nov 2020.

[28] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105, 2007.

[29] Gustavo A. Oliva, Ahmed E. Hassan, and Zhen Ming Jiang. An exploratory study of smart contracts in the ethereum blockchain platform. *Empirical Software Engineering*, 25(3):1864–1904, May 2020.

[30] G. A. Pierro and Henrique Rocha. The influence factors on ethereum transaction fees. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 24–31. IEEE, 2019.

[31] E. L. Sidorenko. The legal status of cryptocurrencies in the russian federation. *Economics, taxes and law*, 11(2):129–137, Nov 2018.

[32] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 9–16, 2018.

[33] S. Ducasse, H. Rocha, S. Bragagnolo, M. Denker, and C. Francomme. SmartAnvil: Open-Source Tool Suite for Smart Contract Analysis. In *Blockchain and Web 3.0: Social, economic, and technological challenges*. Routledge, February 2019.

[34] R. Tonelli and G. A. Pierro. Paso: A web-based parser for solidity language analysis. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 16–21, 2020.