Code Cloning in Smart Contracts on the Ethereum Platform: An Extended Replication Study

Faizan Khan*, Istvan David*, Daniel Varro, Shane McIntosh

Abstract—Smart contracts are programs deployed on blockchains that run upon meeting predetermined conditions. Once deployed, smart contracts are immutable, thus, defects in the deployed code cannot be fixed. As a consequence, software engineering anti-patterns, such as code cloning, pose a threat to code quality and security if unnoticed before deployment. In this paper, we report on the cloning practices of the Ethereum blockchain platform by analyzing 33,073 smart contracts amounting to over 4MLOC. Prior work reported an unusually high 79.2% of code clones in Ethereum smart contracts. We replicate this study at the conceptual level, i.e., we answer the same research questions by employing different methods. In particular, we analyze clones at the granularity of functions instead of code files, thereby providing a more fine-grained estimate of the clone ratio. Furthermore, we analyze more complex clone types, allowing for a richer analysis of cloning cases. To achieve this finer granularity of cloning analysis, we rely on the NiCad clone detection tool and extend it with support for Solidity, the programming language of the Ethereum platform. Our analysis shows that most findings of the original study hold at the finer granularity of our study as well; but also sheds light on some differences, and contributes new findings. Most notably, we report a 30.13% overall clone ratio, out of which 27.03% are exact duplicates. Our findings motivate improving the reuse mechanisms of Solidity, and in a broader context, of programming languages used for the development of smart contracts. Tool builders and language engineers can use this paper in the design and development of such reuse mechanisms. Business stakeholders can use this paper to better assess the security risks and technical outlooks of blockchain platforms.

Index Terms—Code cloning, Smart contracts, Ethereum, Blockchain, Empirical Study

1 Introduction

Blockchains offer a novel computation paradigm for distributed systems, where information has to be stored without the possibility of modification. Smart contracts are software programs deployed on a blockchain. Once deployed, these programs are immutable and execute for as long as the platform is active. Due to its immutable nature, repair in the deployed code is not possible as the modified source code needs to be redeployed as a new instance. As a consequence, bad software engineering practices pose more severe threats in blockchains than in traditional software settings. Vulnerabilities in smart contracts can result in substantial financial repercussions, as the majority of deployed smart contracts are written for financial applications [1].

A particular bad practice that can deteriorate many functional and extra-functional properties (e.g., security, reliability, and performance) of a software system is the abundance of duplicated source code, also known as code cloning. Code clones can be commonly found in software systems. Studies show that a large proportion of code in software projects (6% to 50%) is duplicated [2], [3]. Although some benefits of code cloning exist—such as improved learning curve of APIs and rapid bug workarounds [4]—unintentional cloning [5] affects the quality [6] and maintainability [7] of source code adversely. For example, upon the detection of a bug in a clone, its copies must be checked for bugs as well. Such problems are further exacerbated by code similarity often going beyond simple copy-and-paste [8], rendering the management of clones a complex problem.

Studies show that clones are ideal targets for refactoring aimed at improving the design of the software [9]. Despite the apparent benefits, the use of clone detection tools is limited in the development of smart contracts. This is partly attributed to the fact that the majority of clone detection tools are designed for traditional programming languages [10], and only limited support exists for the novel class of programming languages targeting decentralized execution platforms, such as blockchains. As a consequence, the vast body of knowledge on clone detection in traditional programming languages, such as C, C++, and Java [3], [11], cannot be exploited in programming languages used for developing smart contracts, such as Solidity for Ethereum [12].

In this paper, we report on the cloning practices on Ethereum,[1] one of the most frequently used blockchain platforms. We have designed and carried out a study to analyze 33,073 smart contracts, containing more than four million lines of code (MLOC).

- *F. Khan and D. Varro are with the Department of Electrical and Computer Engineering, McGill University, Canada. E-mail: faizan.khan3@mail.mcgill.ca, daniel.varro@mcgill.ca*
- *I. David is with the Department of Computer Science and Operations Research, Université de Montréal, Canada. E-mail: istvan.david@umontreal.ca*
- *S. McIntosh is with the David R. Cheriton School of Computer Science, University of Waterloo, Canada. E-mail: shane.mcintosh@uwaterloo.ca*
- * *F. Khan and I. David contributed to this paper equally.*

[1] https://ethereum.org/en/

Prior work by Kondo et al. [13], reported an unusually high 79.2% proportion of code clones on the Ethereum platform. Our work is an *extended conceptual replication* of [13], that is, we (i) pose the same research questions; but (ii) use different methods to answer them; and by that, (iii) refine and extend the findings of the original study.[2]

Specifically, in our study, we analyze code cloning practices *at the level of function blocks*, as opposed to the file-level analysis of the original study. To achieve this finer granularity of cloning analysis, we opt for the NiCad clone detection tool [14] instead of Deckard [15] which was used in the original study, and extend it to support Solidity, the programming language of the Ethereum platform. NiCad has been frequently used for clone detection tasks in conventional software systems. It has been thoroughly analyzed and benchmarked in previous studies to identify optimal configuration settings for detecting clones [16]. We also extend the scope of potential clone types to better identify near-miss (Type-3) clones, which can detect clones with modifications such as changed, added, or removed statements [17]. We assess the ratio of clones in the code base by removing clone duplicates, i.e., clones that have been identified multiple times as instances of different clone types. This allows for a better understanding of the types of cloning-related issues in Solidity smart contracts [18]. This step is explained in detail in Section 5.2.1.

To the best of our knowledge, this paper is the first to explore cloning in Solidity smart contracts at this finer granularity and with an awareness of these types of clones.

Results. We corroborate many findings of Kondo et al. [13], but observe some important differences as well. Most importantly, we observe that the clone ratio decreases from 79.2% to 30.13% at the finer level of granularity of functions. Moreover, we observe that the vast majority of clones (90%) are Type-1 clones (i.e., exact replicas). This 90% proportion among the clone types tends to be steady over an extended period of time, while the total number of clones increases; i.e., smart contract development practices heavily rely on copy-and-paste mechanisms. Tool builders and language engineers can use these results to improve reuse mechanisms in smart contract programming languages, including, but not limited to Solidity. Business stakeholders can use this paper to better assess the security risks and technical outlooks of blockchain platforms.

Fostering replication. The replication package containing the data and analysis scripts of our study are publicly available for the independent verification or replication.[3]

2 Background

Clone detection aims to identify repeated code. Clones are identified based on a similarity relation between their two respective code fragments. A *clone fragment* is a sequence of contiguous lines of code that is similar to another, non-overlapping sequence of contiguous lines of code. Clones with similar properties form a *clone pair*, and when there are many similar clones, they form a *clone class* (also referred to as *clone group* or *clone cluster*) [16].

[2]In the remainder of this paper, we refer to [13] as the *original study*.
[3]https://zenodo.org/record/6975351

Clone granularity can be either *free* or *fixed*. Free granularity clone detection considers the source code as a whole and does not make use of syntactic boundaries, such as functions, blocks, or statements [10]. Fixed granularity, however, incorporates such syntactic units. As such, fixed granularity provides a more precise estimate of clone ratio, and is more useful than free granularity in the eventual refactoring of the duplicated code [19]. Furthermore, clone detectors of free granularity produce a higher number of false positives [11], [20], which are code fragments that have been cloned with a purpose, such as getter/setter methods in Java code. In this paper, we use a fixed granularity at the function level.

Syntactic clones are identified based on textual program code, while the identification of semantic clones requires an analysis of the behavior of the units of code [21]. In this paper, we focus on syntactic clones, which are further divided into three types. *Type-1* clone fragments are exactly identical except for variations in whitespaces, layout, and comments. For example, Listings 1 and 2 would be Type-1 clones of each other, were their respective source code on Lines 5 and 7 identical. *Type-2* clone fragments include Type-1 clones, but allow for differences in identifiers, literals, and data types. For example, Listings 1 and 2 would be Type-2 clones of each other, were the respective assigned values on Lines 5 and 7 identical. *Type-3* clone fragments include Type-2 clones, but allow code fragments to differ in complete lines of code, thereby capturing clones with entire lines added or removed. The number of lines to be tolerated is defined by the *dissimilarity threshold*, in ratio with the overall code block. In our experiments, we set the dissimilarity threshold to 0.3, which classifies clones as Type-3 if at least 70% of the normalized subsequences match. Accordingly, Listings 1 and 2 are Type-3 clones. They differ on two out of twenty lines, i.e., a $\frac{2}{20}$ = 0:1 dissimilarity or 90% similarity, which exceeds the threshold of 70%. Identifiers in Type-2 and Type-3 clones are normalized by performing a renaming strategy. The two most common renaming strategies are *blind renaming*, where all identifiers are replaced with the same key; and *consistent renaming*, where identifiers are given a unique key. For example, the line `int sum = 0` is changed to `x x = 0` by blind renaming, and to `x1 x2 = 0` by consistent renaming. Line 5 in Listings 1 and 2 is changed to `x = "MT"` and `x = "NEM"`, respectively by both blind and consistent renaming. Were the variables named differently, e.g., `symbol = "MT"` in Listing 1 and `sym = "NEM"` in Listing 2, blind renaming would still change them to `x = "MT"` and `x = "NEM"`; however, consistent renaming would change them to `x1 = "MT"` and `x2 = "NEM"`.

In this paper, we identify Type-1, Type-2, and Type-3 clones. The latter two types are further refined into blindly and consistently renamed clones (Type-2b, Type-2c; and Type-3b, Type-3c; respectively). Semantic clones (Type-4) are beyond the scope of this paper.

Smart contracts are programs that can be reliably executed by a network of anonymous distributed nodes without the need for a centralized trusted authority. The collection of these nodes forms a distributed computing platform called a *blockchain* [22], upon which smart contracts are executed. The name blockchain reflects the fact that transactions (i.e., actions initiated by an externally-owned account, such as

Listing 1: The MT.sol smart contract.

```
contract MT is ERC20Interface, SafeMath {
  ...
  constructor (string memory _name) public {
    name = _name;
    symbol = "MT";
    decimals = 18;
    totalSupply = 500000000000000000000000000;
    balanceOf[ msg.sender ] = totalSupply;
  }

  function transfer( address _to, uint256 _value) public
      returns (bool success) {
    require (_to != address (0));
    require (balanceOf[ msg.sender ] >= _value);
    require (balanceOf[ _to] + _value >= balanceOf[ _to]);
    balanceOf[ msg.sender ] =SafeMath.safeSub(balanceOf[ msg.
        sender ],_value)
    balanceOf[_to] =SafeMath.safeAdd(balanceOf[_to],_value)
    emit Transfer( msg.sender , _to, _value);
    return true ;
  }
  ...
}
```

Listing 2: The NEM.sol smart contract.

```
contract NEM is ERC20Interface, SafeMath {
  ...
  constructor (string memory _name) public {
    name = _name;
    symbol = "NEM";
    decimals = 18;
    totalSupply = 860000000000000000000000000;
    balanceOf[ msg.sender ] = totalSupply;
  }

  function transfer( address _to, uint256 _value) public
      returns (bool success) {
    require (_to != address (0));
    require (balanceOf[ msg.sender ] >= _value);
    require (balanceOf[ _to] + _value >= balanceOf[ _to]);
    balanceOf[ msg.sender ] =SafeMath.safeSub(balanceOf[ msg.
        sender ],_value)
    balanceOf[_to] =SafeMath.safeAdd(balanceOf[_to],_value)
    emit Transfer( msg.sender , _to, _value);
    return true ;
  }
  ...
}
```

a human) within this network are stored in a chain of immutable blocks. One commonly used platform is Ethereum [12]. Solidity is an object-oriented and statically-typed programming language designed for developing smart contracts, in uenced by C++, Python, and ECMAScript.

Listings 1 and 2 show code snippets from the MonPay-Token (MT.sol)[4] and NEM token[5] smart contracts written in Solidity. Both smart contracts create a custom token that can be treated as a virtual currency. The listings show that the contracts are identical apart from their symbols and the total supply of tokens. Both of the smart contracts use the SafeMath contract and ERC-20 interface[6] to implement the Tokenfunctionality. There are 20 instances of the same smart contract being repeated with small changes in our corpus. Such repetitions pose a threat to the platform, as vulnerabilities in any of these base smart contracts would potentially affect a large number of smart contracts in production.

3 Summary of the Original Study

Kondo et al. [13] report (i) the amount of cloned Solidity smart contracts on the Ethereum platform; (ii) the characteristics of clones; and (iii) the overlaps of clones with code blocks of smart contract libraries (e.g., OpenZeppelin). The authors analyzed 33,073 smart contracts amounting to about 4 MLOC, and 13 releases of OpenZeppelin[7] to answer three research questions.

3.1 Research questions and major findings

The research questions and key observations from the original study are the following.

RQ1. How frequently are veri ed contracts cloned?

79.2% of the studied contracts are clones. In particular: 16.7% of the studied contracts are Type-1 clones; 43.3% of the studied contracts are Type-2 clones. Type-3 clones were considered out of the scope due to their detection still being actively researched.

RQ2. What are the characteristics of clusters of similar veri ed contracts?

The original study reports on three inferred characteristics: (i) category, (ii) activity concentration, and (iii) authorship. In particular: (i) 9 out of the top-10 largest clusters are token managers; (ii) transaction activity tends to be concentrated on a few contracts; and (iii) contracts in a cluster tend to be created by many authors.

RQ3. How frequently code blocks of veri ed contracts are identical to those from OpenZeppelin?

About one-third of all 165,005 code blocks extracted from veri ed contracts are identical to OpenZeppelin code blocks. 36.3% of the veri ed contracts include at least one code block that is identical to an OpenZeppelin code block. 50% of the code blocks from 26.3% of the veri ed contracts are identical to OpenZeppelin code blocks. The ERC-20 OpenZeppelin category is the most frequently reused category, containing code blocks to support the implementation of token contracts that comply with the ERC-20 standard. SafeMath.sol is the most frequently reused OpenZeppelin code le, containing functions that perform mathematical operations ef ciently and safely.

3.2 Approach

Clone granularity and detection tool. Deckard [15], a free granularity clone detector was used to detect clones between Solidity code les.

Clone types considered. Type-1 and Type-2 clones were considered as part of RQ1.

Corpus. The corpus consists of 4,004,543 lines of code, extracted from 33,073 veri ed smart contracts. The les were retrieved from Etherscan[8] in July 2018. Etherscan is an analytics platform for the Ethereum blockchain that analyzes each block on Ethereum and provides insights on each deployed contract. The existence of source code on Etherscan indicates that the source code in Solidity provided by Etherscan matches the bytecode deployed to Ethereum,

[4] etherscan.io/token/0xa0b469450e78b3a85d828d454696f8e4bd420038
[5] etherscan.io/token/0xc14db8e15690c28752dbda133f51821402d29f29
[6] https://eips.ethereum.org/EIPS/eip-20
[7] https://github.com/OpenZeppelin/openzeppelin-contracts
[8] https://etherscan.io

and therefore, it is considered verified. Thus, the corpus contains only verified contracts. Verified smart contracts publish their flattened version on Etherscan. This flattened version of the source code is referred to as the code leaf of a verified contract. No restriction on the transaction number on the contracts was imposed. The corpus was compared with 13 releases of OpenZeppelin in RQ3, released between 2016-11-24 and 2018-08-10, with continuous growth in the size of 1–5 KLOC over time.

4 Study Design

In this section, we discuss the design of our replication study, following the guidelines of Carver [23].

4.1 Type of replication

We have carried out a conceptual replication study [24]. That is, we test the same research questions on the same corpus, but use different measures and techniques.

4.2 Motivation for replication

Our work is motivated by the high clone ratio in Solidity smart contracts reported by the original study being significantly higher than clone ratios in traditional software systems. The figures are suggestive of systemic issues in the design and methodology of engineering Solidity smart contracts. Other work [25], [26] confirms this unusually high rate of clones. Such unusual figures have to be verified by independent studies, especially because (i) the cost of performing a transaction or executing a smart contract is proportional to its size, [9] and thus, minimizing the size of smart contracts can result in direct cost reduction; and (ii) the majority of smart contracts are deployed in financial applications, and thus, vulnerabilities might have serious financial repercussions [27]. Furthermore, the approach of the original study is often prone to false positives due to the free clone granularity it relies on (see Section 2). Therefore, we set out to replicate the analyses of the original study using a fixed granularity at the function level. We conjecture that this new viewpoint from which cloning can be observed also enhances the applicability of the results in refactoring processes aiming to eliminate duplicated code.

4.3 Level of interaction with the original researchers

Ours is an external replication, i.e., the original researchers were not involved in the replication [23]. The interaction with the original researchers was restricted to inquiring about the study's data and receiving the data package along with technical pointers regarding its structure.

4.4 Changes to the original study

Clone granularity and detection tool. We approach clone detection with a fixed granularity, fixing our scope at the function level. Due to the lack of support for fixed granularity analysis by the clone detection tool used in the original study (Deckard [15]), we rely on the NiCad clone detection tool [14]. NiCad does not support Solidity out-of-the-box. Therefore, we contribute a custom Solidity grammar, [10] which makes our analysis and other future work possible.

Clone types considered. In addition to the Type-1 and Type-2 clones that the original study reports on, we also include Type-3 clones in our scope. Furthermore, to refine our reporting, we (i) split Type-2 and Type-3 clones into subtypes based on the renaming strategy that has been applied in the specific clone detecting case; and (ii) provide a systematic process to remove duplicated clones.

5 Experimental Setup

In this section, we present our experimental setup. As shown in Figure 1, our study is composed of three phases.

5.1 Tool configuration and clone detection

In this phase, we select the clone detector for our study and configure it (Section 5.1.1), develop a grammar to support clone detection in Solidity smart contracts (Section 5.1.2), carry out the clone detection (Section 5.1.3), and download the releases of OpenZeppelin to be analyzed (Section 5.1.4).

5.1.1 Tool selection and configuration

We set out to select a clone detection tool that was (i) freely available and (ii) customizable. While there exists a long list of freely available clone detection tools [16], we found NiCad being easily customizable for our purposes. NiCad is a text-based clone detection tool that was primarily designed to detect near-miss clones. It has been widely used for clone detection studies, thanks to its high precision and high recall for detecting near-miss clones [16], [28]. Following the suggestions of Wang et al. [29] and the settings used by Hasanain et al. [3], we set the granularity threshold to 10 LOC and the dissimilarity threshold for Type-3 clones to 0.3. These are also the default settings of NiCad.

5.1.2 Grammar development

To conduct our experiment, we extended NiCad with a grammar to enable the parsing of Solidity source code. Our grammar [10] is inspired by the grammar for Solidity available in ANTLR. [11] In order to extract a parse tree, NiCad expects a context-free grammar for the source-code language to be provided in a TXL grammar format [30]. TXL is a programming language for rule-based transformations. The TXL grammar not only provides the correct input for parsing, but also provides special markers, such as indent, extent, and newlines for pretty-printing the source code.

5.1.3 Clone detection

The clone detection of NiCad consists of (i) parsing and extraction of potential clones, (ii) pretty-printing and normalizing, and (iii) clone clustering.

Parsing and extraction of potential clones. NiCad extracts a parse tree representation from the source code, filters out

[9] See the documentation of Gas, the unit of computational effort required to execute operations on the Ethereum platform at https://ethereum.org/en/developers/docs/gas/.

[10] The grammar is available at https://github.com/eff-kay/nicad6.

[11] https://github.com/antlr/grammars-v4/tree/master/solidity

Figure 1: Overview of the study.

irrelevant blocks, performs normalizations, and transforms the parse tree back to source code. We developed pretty-printers for the grammar to ensure that all functions are evaluated consistently. The basic rules of pretty-printing are the following: (i) function signatures appear on a single line; (ii) block parentheses follow the ECMAScript standard; and (iii) every complete statement appears on its own line. A block of at least ten lines of normalized source code is considered for regular clones because according to previous studies, this is the best threshold value for the NiCad tool to detect code clones from Java and C source code [29]. Most studies of clone detection consider code clones of less than ve LOC to be false positives [20] or micro-clones [31], [32], [33] – an entirely different type of copy-pasted artifact.

Flexible pretty-printing and normalization. In addition to pretty-printing, NiCad is capable of context-sensitive normalizations, i.e., normalization based on the context of the code fragment. For this initial exploration, we use the default normalization settings of NiCad. NiCad detects clone pairs in this step by performing a line-wise comparison of the normalized code snippets.

Clone clustering. Finally, NiCad conducts a basic cluster analysis of the clones identi ed to combine similar clone fragments into the same clone cluster. Clones in the same cluster belong to the same clone class.

Corpus. We use the corpus of the original study, [12] described in detail in Section 3.2. The corpus contains 33,073 veri ed smart contracts, amounting to 4,004,543 lines of code. By using veri ed contracts deployed to Ethereum, we can be sure that the corpus is representative of code in production.

5.1.4 OpenZeppelin code analysis

We download the contracts of the twelve releases of OpenZeppelin that were analyzed by Kondo et al. [13]. We use NiCad to extract contract and function blocks from the corpus as well as from the OpenZeppelin releases. As explained in Section 5.1.3, the extraction of contracts by NiCad normalizes the source code within each code-block. Then, we calculate unique hashes for every code block extracted from OpenZeppelin releases. We will compare these hashes with the hashes extracted from the corpus.

[12] https://github.com/SAILResearch/suppmaterial-18-masanari-smart_contract_cloning

5.2 Metadata extraction and preprocessing

In this phase, we preprocess the clone detection results by removing duplicated clones (Section 5.2.1), extracting metadata (Section 5.2.2), and preparing the data (Section 5.2.3) for the subsequent analysis.

5.2.1 Duplicate removal

Due to the overlapping de nition of clone types, some clones might belong to multiple clone classes conforming to different types [18]. For example, if two code fragments are identical, they will also be identical after a blind renaming procedure is performed on them. Consequently, the class of Type-2 clone instances that have been obtained by blind renaming, will contain fragments that are also within the class of Type-1 clone instances. (See Section 2.) We refer to this implied containment relationship between clone classes as the *strictness* of a clone class. Class C_t of clone instances of type t is stricter than $C_{t'}$ if each clone that belongs to C_t also belongs to $C_{t'}$. It is directly implied by this de nition, that $C_t \quad C_{t'}$. We construct the classes of our approach based on (i) the type of contained clone instances, and the renaming procedure. For simplicity, we refer to these classes by their type and by appending c (consistent) or b (blind) to the name, depending on the renaming procedure that was applied while extracting the clones. Equation 1 de nes the relations between the resulting clone classes.

$$\text{Type-1} \quad \text{Type-2c} \quad \text{Type-2b} \quad \text{Type-3} \quad \text{Type-3c} \quad \text{Type-3b} \tag{1}$$

We use this hierarchy to remove clone duplicates. The process iterates through the sets from the strictest to the weakest, and excludes every clone present in the current set from the sets that are weaker than the current set. That is, rst, we exclude every Type-1 clone from classes Type-2c, Type-2b, etc; then, we exclude every Type-2c clone from classes Type-2b, Type-3, etc; and as the last step, we exclude every Type-3c clone from class Type-3b.

5.2.2 Metadata extraction

For the 33,073 veri ed smart contracts in the corpus, we have collected additional metainformation from the Etherscan[8] analytics platform. We collect two types of information: *Creation dates* to answer the clone evolution aspect of RQ1, and *Author information* to answer the authorship aspect of RQ2. Both information is extracted from the transaction log of contracts. In about 3.5% of contracts, the creation date was not available from Etherscan. Those cases are excluded from the analyses. We also calculate the length of les in this phase for further analysis.

5.2.3 Data preprocessing

To allow for fast analysis in the subsequent phase, we take care of the computation-intensive tasks of (i) merging metainformation with the data obtained from the clone analysis, and (ii) preprocessing the merged data in various ways. For example, we calculate quarterly gures for RQ1 and calculate Gini-coef cients for RQ2. The preprocessing scripts are available from the replication package. [3]

5.3 Analysis and reporting

In this phase, we analyze the data (Section 5.3.1) and carry out the comparison with the original study (Section 5.3.2). Finally, we report our ndings (Section 5.3.3).

5.3.1 Analysis

We analyze cloning patterns quantitatively. For each research question, we design an analysis and encode it in automated data analysis scripts in Python. The scripts are available from the replication package. [3]

5.3.2 Comparison

We compare our results with the original study by Kondo et al. [13] by research question. We map our ndings (Section 6) to the fourteen observations of the original study and observe whether our ndings corroborate the speci c observations (Table 5).

5.3.3 Reporting

Finally, we conduct a narrative synthesis [34] to synthesize the main ndings from the analyses. We conducted multiple discussions on the ndings to formulate conjectures and hypotheses. We were especially interested in identifying feasible and actionable directions for the Ethereum/Solidity community, and for blockchain communities in general. These discussions are reported at the end of the each subsection in Sections 6.1–6.3.

6 Empirical study on code cloning

In this section, we present the results of our study by answering the three research questions.

6.1 RQ1: How frequently are veri ed contracts cloned?

6.1.1 Approach

To conduct this experiment, we calculate the clone percentage as the ratio of the total normalized LOC of clone clusters, and the total normalized LOC, as shown in Equation 2.

$$\text{Clone\%} = \frac{\text{NormLOC}_{\text{cloned}}}{\text{NormLOC}_{\text{total}}} \quad 100 \qquad (2)$$

where $\text{NormLOC}_{\text{cloned}}$ is the sum of all cloned lines after the lines are normalized and the repeated clones are removed. To calculate $\text{NormLOC}_{\text{cloned}}$ for a clone cluster, we count the total number of fragments in the clone cluster and multiply it by the number of lines.

6.1.2 Findings

Clone ratio. We have observed that 30.13% of the corpus are clones. As shown in Table 1, 27.03% are Type-1 clones; 2.05% are Type-3 clones; while Type-2b, Type-2c, Type-3b and Type-3c amount to 1.05%. Our experiments show that 89.7% of all clones (27.03% of the corpus) are of Type-1, i.e., exact clones.

Figure 2: Relationship between the proportions of clones and contracts with two characteristic values highlighted.

Table 1: Clone proportions.

Clone type	Proportion
Type-1	27.03%
Type-3	2.05%
Type-2c	0.54%
Type-3b	0.33%
Type-3c	0.15%
Type-2b	0.03%
total cloned	30.13%
clone-free	69.87%

Clone clusters. A small proportion of clone clusters encompass a large proportion of clones. As shown in Figure 2, 20% of all clusters encompass 71.9% of all clones; and half of the clones can be found in just 2.07% of clusters.

Clone evolution. The number of clones among newly created contracts continues to increase over time. The amount of non-Type-1 clones increases proportionally within the cloned code base. The number of all clones has doubled quarterly from early 2016 to early 2018, as shown in Figure 3a. However, the proportion of clones other than Type-1 remained steady after the initial uptick in Q2 2016, as shown in Figure 3b. That is, the increasing number of code clones in Figure 3a are predominantly Type-1.

6.1.3 Discussion

We nd that the proportion of clones in smart contracts is 30.13%. This gure is on par with the ones reported by studies on conventional software systems [2], [35], [9]. However, an important difference with conventional software systems is that the source code of deployed blockchain applications is immutable and therefore, modi cations, such as bug xes, are not possible after deployment. This, in turn, ampli es the threat of exploiting vulnerabilities that spread across the code base by cloning. This mechanism has been demonstrated, e.g., in the Parity Wallet Hack [36], in which a malicious agent drained 153,037 ETH (over 428 million USD at the time of writing) from three high-pro le contracts. However, we also observe that most of the clones in smart contracts are of Type-1 (27.03% of the overall corpus and 89.7% of all clones), that is, the majority of the functions are being copied without any modi cations. As Tsantalis et al. [9] report, Type-1 clones are easier to refactor using existing clone refactoring tools, which suggests that refactoring the code base of the Ethereum blockchain platform

(a) Number of newly created clones.

(b) Number of newly created non-Type-1 clones.

Figure 3: Evolution of clone numbers and percentages.

might be feasible. Furthermore, the analysis of clone clusters suggests that there are hotspots of cloned source code that should be the primary targets of refactoring. Refactorings related to inheritance—such as class and method extraction, method pull up and push down—could be of particular utility. While inheritance is a supported language feature in Solidity, it is apparently underutilized, as evidenced by the high proportion of clones despite the immutability of the deployed code and demonstrated in Listings 1-2. This might indicate a need for better tool assistance in recognizing abstraction/inheritance opportunities.

6.2 RQ2: What are the characteristics of clusters of similar verified contracts?

6.2.1 Approach

We extract the function identifier within each clone fragment using a custom regular expression. For Type-1 and Type-3 clone clusters with no renaming, the function identifiers are the same for each clone fragment within a clone cluster. Thus, there is one function identifier per clone cluster. For the rest of the clone types, the unique function identifiers are extracted from all clone fragments within a clone cluster.

Bartoletti and Pompianu [1] studied 811 smart contracts written in Solidity and categorized them by the design patterns that they apply. We use the same categorization applied at the function level. In addition, we add a new category of Helper functions, which broadly includes all functions not categorized elsewhere. Below, we describe the three categories that are most relevant to our study.

Token. The code clones in these patterns are used for the distribution of tokens or fungible goods to users. Token is an abstract concept that can represent anything that is countable and transferable, e.g., shares in a company, outcomes of an event, etc. DigixGold[13] is an instance of the Token pattern, which tracks the ownership of a fixed amount of gold by using Tokens. A subset of token managers are authorization contracts. The function of the code clones in this category is to evaluate the authorization of the caller. With any transaction, one needs to check whether the parties invoking the transactions are permitted to do so.

Oracle. Some smart contracts require data from outside the scope of the blockchain, e.g., to determine the latest posted exchange rate for the US dollar. For this purpose, a collection of special smart contracts, called Oracles have been created for the Ethereum platform. Oracle smart contracts provide hooks to the outside world, which allows an external service to update the state of the Oracle. This allows Oracles to act as stable interfaces between the outside world and other smart contracts. In practice, a non-Oracle smart contract queries the Oracle smart contract, instead of querying an external service. On the other hand, an external service sends a transaction to the Oracle when an update to the data encapsulated by the Oracle is requested.

Helper functions. This category includes functions that serve as wrappers around existing functions. Smart contract-specific functions, such as initialize and migrate also belong to this category. The category can be considered as a collection of functions not categorized elsewhere.

[13] https://digix.global/#/

Table 2: Top 20 most frequently cloned functions.

	Function ID	Clone count	Category
1	transferFrom	748	Token
2	transfer	610	Token
3	() payable noAnyReentrancy	512	Token
4	buyTokens	398	Token
5	_transfer	163	Token
6	buy	140	Token
7	Crowdsale	106	Helper
8	createTokens	98	Token
9	withdraw	98	Token
10	sell	97	Token
11	purchase	81	Token
12	decreaseApproval	78	Token
13	mint	72	Token
14	claimTokens	69	Token
15	finalize	69	Helper
16	refund	64	Token
17	deploy	61	Token
18	tokensOfOwner	57	Token
19	__callback	54	Oracle
20	investInternal	46	Token

6.2.2 Findings

Cloned functionality. Table 2 lists the 20 functions of the code base that contain the most clones. 17 of the 20 contracts are Token-related, i.e., they provide functionality for the management and provision of contacts, such as buy, sell, withdraw, refund, etc. These functions have the same intent as the transfer and transferFrom functions. Another group of functions, including createTokens, mint, and deploy, are all variants of a mechanism for increasing the supply of Tokens available for a smart contract.

The second most frequent category are Helpers. Crowdsale is a common Helper function that is used to set the initial conditions for carrying out a crowdsale operation. Crowdsale is a method of flash sale where a number of tokens are allocated to be sold within a time window. The presence of this function is not unexpected, as a large number of

(a) Gini-coef cient of clone clusters (with at least 10 clones).

(b) Relative rank of the most active contract.

Figure 4: Clone clusters (with at least 10 clones) and the activity of the related contracts.

smart contracts are written to conduct Initial Coin Offerings for raising capital for projects. Similarly, finalize is another Helper function, with the purpose of terminating the crowdsale initiated by the previous function.

One of the top 20 categories is the Oracle functionality, speci cally, the __callback function. As the name suggests, the purpose of this function is to serve as a callback function to be invoked when an external query is completed. In our example, the most common calls are made to the Oracle smart contract. As explained in Section 6.2.1, an Oracle serves as a doorway between the blockchain and the external world. Therefore, the purpose of queries to Oracles is to access external data and resources.

Activity. Following the original study, we measured activity in terms of the number of transactions that are related to contracts. First, we observe that activity tends to be concentrated on a few contracts. We use the Gini-coef cient [37] as the measure of inequality among transactions related to clone clusters. A Gini-coef cient of 0 indicates no inequality among values, while a value of 1 indicates maximal inequality. To obtain meaningful results, we investigate clusters with at least ten clones. As Table 3 and Figure 4a show, the overall Gini-coef cient of the clone clusters is 0.86. Second, the medians in Figure 4b show that in 50% of the cases, the most active contract was created before 66.7% of contracts in the same clone cluster. (The proportion above the median line in the Overall case is 66.7%.) This number is higher in Type-2c clones, where 50% of affected contracts were created before 90.91% of the rest of the cluster.

Table 3: Gini-coef cients.

Clone type	Proportion
Type-1	0.86
Type-2c	0.73
Type-3	0.86
Type-3b	0.77
Overall	0.86

Figure 5: Distribution of authorship entropy with the median entropy highlighted.

Authorship. Contracts in a clone cluster tend to be created by many authors. We measure this observation by the normalized Shannon-entropy [38] within a cluster. Maximum entropy (1.0) is measured for distributions with elements of uniform probability, i.e., in clone clusters with contracts that have equal transactions. The less uniform the probabilities of elements in a distribution, the lower the entropy. To obtain meaningful results, we once again investigate clusters with at least ten clones. As Figure 5 shows, the median entropy in our sample is 0.7, while the median normalized cluster size is close to zero (0.058). This means the average cluster is relatively small compared to the largest clusters while showing high entropy, i.e., a large variance in the authors. The darker area in the bottom-left corner shows that the majority of clone clusters have high entropy.

6.2.3 Discussion

Out of the functionality that is subject to frequent cloning, token management contracts, including authorization, pose the most pressing issue. A detailed look at the cloned functions reveal that basic transaction functions such as transfer and createTokens are among most frequently cloned. Providing a library of secure transfer primitives could simplify the development of such functionality. From a language design point of view, declarative and veri able language constructs have been identi ed as potential enablers to a more secure design of smart contracts [39]. The bene ts of such techniques have been demonstrated in blockchain languages, such as Pact and Liquidity.

The relatively high Gini-coef cients suggest that activity within a cluster tends to focus on a small number of contracts. The overall Gini-coef cient of 0.86 is roughly equivalent to a cluster of ten contracts with nine contracts having one transaction, and one contract having 250 transactions. Vulnerabilities in such frequently used contracts are more likely to be identi ed by malicious attackers. The effects of such vulnerabilities, in turn, are ampli ed by code

Table 4: Top 10 cloned functions from OpenZeppelin.

	Function	Clone count	% of all	Contract
1	transferFrom public returns (bool)	15,287	28.43	StandardToken
2	decreaseApproval public returns (bool)	12,021	22.35	StandardToken
3	transfer	11,951	22.22	BasicToken
4	allowance	1,602	2.98	StandardToken
5	approve	974	1.81	StandardToken
6	burn	779	1.45	BurnableToken
7	transferFrom returns (bool)	748	1.39	StandardToken
8	decreaseApproval returns (bool success)	663	1.23	StandardToken
9	_burn	612	1.14	BurnableToken
10	TokenVesting	540	1.00	TokenVesting

cloning as the same vulnerabilities can be anticipated in the contracts of the same clone cluster.

The high entropy in authorship suggests that cloning is a widespread phenomenon on Ethereum. Such community-wide bad practices are often addressed by guidelines published by community leaders, such as the Python Enhancement Proposal (PEP) 8 style guidelines for Python [40]. However, such general rules cannot be enforced in a computer-automated fashion, and a better solution could be establishing community-specific DevOps processes that include the usage of quality gates enforced by code quality tools that evaluate contracts that are ready to be deployed.

6.3 RQ3: How frequently are code blocks of verified contracts identical to those from OpenZeppelin?

6.3.1 Approach

To answer the research question, we identify the code blocks present in OpenZeppelin releases that are also present in the corpus. We do so by (i) extracting code blocks from OpenZeppelin, (ii) calculating their hashes (as explained in Section 5.1.4), and (iii) comparing those hashes with the hashes calculated for the code blocks of the corpus.

6.3.2 Findings

Table 4 shows the 10 most commonly cloned functions from OpenZeppelin, along with their category, the respective number of clones, and the proportion of these clones in the overall set of OpenZeppelin (OZ) clones.

Clone proportion. Of all verified contracts, 21.79% have functions identical to those of OpenZeppelin. As seen in Table 4, the three most cloned functions encompass 73% of all clones from OpenZeppelin.

Functionality. Most functions have been defined in the StandardToken OpenZeppelin contract. Six of the ten most cloned functions (Table 4) and fourteen of the hundred most cloned functions belong to this category. Other frequently encountered categories are SafeMath (10) and VestedToken (8). The most frequently cloned functionality is related to transfers, accounting for over 50% of cloned functionality.

6.3.3 Discussion

OpenZeppelin serves as a frequent source of code cloning on the Ethereum blockchain platform. The high volume of cloning from OpenZeppelin suggests that mechanisms for reusing functionality from libraries such as OpenZeppelin could reduce the number of clones, and improve the maintainability of the overall code base. This, in turn, could improve the extra-functional properties of Ethereum, such as security, reliability, and integrity. The functionality cloned from OpenZeppelin tends to concentrate on transfer-related functionality, and mostly from the StandardToken contract.

7 Comparison with the original study

In this section, we provide an overview of how the findings in Section 6 align with the results of the original study of Kondo et al. [13]. The mapping between the two studies is shown in Table 5. For the sake of compactness, we have presented our results in slightly different groups of findings. Below we give a detailed explanation.

7.1 RQ1

We have observed the most important difference between our study and the original study while analyzing RQ1.

Clone ratio. The overall proportion of clones that we detect (30.13%) is considerably smaller than the proportion observed in the original study (79.2%). This difference is due to three factors. First, our analysis is performed at the function-level, which is a finer granularity and provides a larger sample of code units that are subject to cloning. Second, we count every identified clone once, as explained in Section 5.1.3. Third, due to the normalization, the clones that we identify are mainly exact copies, further reducing the number of instances of less strict clone clusters. However, as a common treatment in near-miss clone detectors, we normalize the text of the function blocks using standard ECMA formatting, as explained in Section 5.1, reducing the number of false-negatives, and consequently, potentially increasing the number of identified clones.

We corroborate the high ratio of Type-1 clones but observe a much larger proportion of Type-1 clones among all clones. Our experiments show 89.7% of all clones are of Type-1, as opposed to the 21.1% (16.7% overall) reported by the original study. This can be explained by removing Type-1 clones from Type-2 and Type-3 clusters.

The substantial difference between our results and the original study shows that while 79.2% of contract files might be affected by cloning practices, it is typically only a subset of the encoded functionality that is actually cloned.

Clone clusters. The original study found that 20% of clusters encompass 68% of clones. These figures are nearly

Table 5: Mapping the findings of the current study to the observations of the original study.

Finding: current study	Observations: original study	Comparison
	RQ1	
Clone ratio	Observations 1, 4, 5	Refined – different results
Clone clusters	Observation 2	Corroborated
Clone evolution	Observation 3	Corroborated & refined
	RQ2	
Cloned functionality	Observations 6, 7	Corroborated – minor diff
Activity	Observations 8, 9	Corroborated – minor diff
Authorship	Observations 10, 11	Corroborated
	RQ3	
Clone proportion	Observations 12, 13	Refined – different results
Functionality	Observation 14	Refined – different results

identical to those that we observe: 20% of all clusters encompass 71.9% of all clones; and half of the clones can be found in just 2.07% of clusters. We conclude that our results at a finer level of granularity corroborate the findings of the original study at a coarser level of granularity.

Clone evolution. Since the original study also observed an increasing trend, we conclude that our results corroborate the findings of the original study. However, we point out that different types of clones evolve at different paces. Specifically, the amount of newly created Type-1 clones is the predominant factor behind the increasing trend.

7.2 RQ2

We observed minor differences in RQ2 in terms of the cloned functionality (due to the different levels of granularity of the two studies), and the activity of cloned contracts.

Cloned functionality. The original study reports that nine of the ten most populous clusters are related to Token management. Our finer-grained results also show that nine of the top ten clusters are indeed Token management functions. Moreover, 17 of the top 20 are Token management functions. Unlike the original study, we find that the other top clusters were Helper and Oracle functions rather than Token Lockers. The Token Locker category of the original study covers three specific functionalities: `lock()`, `lockOver()` and `release()`. At the finer level of granularity of functions, however, these functionalities prove to be less frequently cloned than at the contract level.

Activity. The original study also observes that transactions tend to be concentrated on a few contracts, and reports an overall Gini-coefficient of 0.817. We enhance the prior observations by adding that Type-2c clones show a lower Gini-coefficient (0.73). We report slightly different figures regarding the relative creation date of the most active contracts. In 50% of cases, the top-active contract of a cluster was found created before 74.7% of other contracts in the original cluster by the original study, and 63.4% by our finer-grained study. However, this difference is minor.

Authorship. We observe numbers that are almost identical to the ones reported by the original study. This means the authors of cloned smart contract files are the same as the authors of cloned functions. Therefore, identifying developers who are responsible for clones can be achieved either at the function or the contract level with similar results, with potentially different runtime performance.

7.3 RQ3

We observed relevant differences both in the detected clone proportion and the cloned functionality. These differences are due to the finer granularity of our analysis.

Clone proportion. The original study reported that 36.3% of verified contracts have at least one code block identical to an OpenZeppelin code block. Our finer-grained results show that this proportion decreases to 21.79% when analyzing code similarity at the level of functions with at least 10 LOC; increases to 47.21% when analyzing code similarity at the level of functions with at least 5 LOC; and increases to 64.59% when not considering a minimum function length. These proportions are on par with, and in some cases exceed the 6–50% cloning rate reported from traditional engineering domains [2], [3], suggesting security risks.

Functionality. The original study reported that ERC20 is the most frequently cloned category from OpenZeppelin, and that ERC20 is more frequently cloned than its concrete implementation, StandardToken. However, our finer-grained analysis shows that the StandardToken implementation of ERC20 is more frequently cloned than ERC20. This is not unexpected because ERC20 is an interface and as such, it only defines function signatures but no bodies. While ERC20 might be the most cloned contract block, it is the concrete implementations of ERC20 that that contribute the most cloned function blocks

8 Related Work

In this section, we briefly review the related work.

8.1 Empirical studies on smart contracts

Bartoletti and Pompianu [1] conducted a study to analyze top blockchain platforms and their usages. They analyzed 834 smart contracts written for the Ethereum and Bitcoin technologies, and grouped the contracts by application domain and design patterns that were applied. We use the same design patterns as the basis for assigning commonly cloned functions into different categories.

Durieux et al. [41] studied nine automated analysis tools for Solidity. Automated analysis tools can aid developers in meeting required functional and extra-functional qualitative measures, resulting in better performing, safer and more reliable code. The authors conclude that state-of-the-art analysis tools fall short in detecting numerous classes of vulnerabilities, identifying only 40% of the vulnerabilities in a testing corpus, and produce a large number of false positives. These results are corroborated by Ghaleb et al. [42] who investigated the effectiveness of static analysis tools for Solidity smart contracts using bug injection. These results provide evidence that code clones are hotspots of software issues because they facilitate the spread of faulty code, code smells, and anti-patterns. The ineffectiveness of analysis

Table 6: Studies on clone detection in Solidity contracts.

Authors	Granularity	Method/Tool	Clone%
Liu et al. [26]	free	Birthmarks/EClone	N/A
Gao et al.[25]	free	SmartEmbed[14]	90%
Kondo et al.[13]	free	Deckard [15]	79.2%
Our study	fixed	NiCad [14]	30.13%

tools positions clone detection as a viable technique to aid in combating the vulnerability of the code base on Ethereum.

Extra-functional properties of blockchains have been analyzed by Li et al. [43] on security, by Rouhani [44] on performance, by Scherer et al. [45] on scalability, and by Belchior et al. [46] on interoperability. These studies provide evidence of quality-related challenges in blockchain applications, which are further exacerbated by code cloning.

Application scenarios of blockchain technologies have been discussed in numerous domains, such as finance [47], e-governance [48], and healthcare [49], where blockchains are positioned as a core technology of foundational infrastructure. In such contexts, poor quality code that is prone to defects and vulnerabilities (e.g., code that is copied and pasted into contexts in which it was never intended to operate) can have serious repercussions.

8.2 Clone detection in Solidity smart contracts

Table 6 summarizes the studies conducted on clone detection in smart contracts.

The first known exploration of clone detection in Solidity smart contracts was conducted by Liu et al. [26]. Their clone detection approach relies on a custom semantics-preserving representation of smart contract traits, called birthmarks. Code similarity is then determined by calculating the statistical similarity between pairs of contract birthmarks. Their work focuses on the evaluations of the birthmark method in detecting self-injected clones, rather than the actual rate of clones in smart contracts themselves.

The SmartEmbed tool was developed by Gao et al. [25] for detecting clones in smart contracts. The traits of Solidity smart contracts are encoded by code embedding vectors. Clones are identified based on the pair-wise comparison of these vectors. The authors report a clone ratio of 90%, which is substantially higher than the clone rate of traditional software artifacts. The exact precision and recall measures of the tool, however, are not reported.

In contrast, we use NiCad [14] in our study. NiCad has been frequently used for clone detection tasks in conventional software systems. In addition, NiCad has been thoroughly analyzed by previous studies for optimum configuration in detecting clones [16], and assessed from a qualitative perspective [28].

8.3 Clone detection studies for other languages

Table 7 summarizes a list of studies conducted on clone detection for programming languages other than those for smart contracts. The studies approach clone detection with a fixed granularity at function/method level, and therefore, the reported figures are comparable to those of our work.

Table 7: Studies on cloning in conventional software.

Author	LOC	Lang.	Tools	Clone%
Tuzun et al. [50]	97K	Java	CCFinder	33%
Tsantalis et al. [9]	51K-209K	Java	Deckard, NiCad	14.3-81.9%
Lague et al. [2]	15M	C	Datrix	6.4%-7.5%
Baxter et al. [35]	400K	C	custom	12.7%-28%
Hasanain et al. [3]	286K	C	NiCad	49%
van Bladel et al. [11]	3.7K-32.4K	Java	NiCad, IClones	23-39%
Our study	2.7M	Solidity	NiCad	30.13%

Tuzun and Er. [50] conducted an empirical analysis of an industrial system consisting of 677 Java files and 97 KLOC. They observed a clone rate of 22% and found that more than half of the files (360 files) have at least one clone. Similarly, Tsantalis et al. [9] conducted a large-scale empirical study using nine open-source projects analyzed by four different clone detection tools (CCFinder, Deckard, CloneDR, and NiCad). They found that the clone rate in test code was between 14.3% and 81.9%, while the clone rate in application code was between 18.1% and 85.7%. In addition, they reported that Type-1 clones can be refactored more easily than other types of clones. This observation aligns well with our finding that about 90% of all clones in Solidity smart contracts are of Type-1, and suggests that the majority of cloning-related code quality issues in Solidity smart contracts can be efficiently refactored.

Laguë et al. [2] conducted a study to reveal the benefits of using clone detection in industrial software development processes. They analyzed a large telecommunication system of 15 MLOC over three years. The results show that the rate of clones stays at a constant level over time, ranging between 6.4% and 7.5%. Similarly, Baxter et al. [35] conducted a clone detection study to identify duplicated code in an industrial system written in C. The size of the code was 400 KLOC. They observed cloning rates of 12.7% clones overall and 28% in three specific subsystems. Their results show that the clone rates may vary for different subsystems.

More recently, Hasanain et al. [3] used NiCad to study clone detection in a large industrial test suite. They found 49% of the code to be duplicated with Type-3 being the prevalent clone type. Van Bladel and Demeyer [11] carried out a similar study on test code in open source projects. They used four different clone detection tools i.e. Nicad, CPD, IClones, and TCore. They report a clone density between 23% and 29% with Type-2 clones having a higher representation. Test suites have not been analyzed in the context of smart contracts, but could provide further valuable insights.

9 Threats to validity

Construct validity. Our observations may be artifacts of the NiCad configuration settings that we used, rather than meaningful observations about cloning tendencies in Solidity smart contracts. To combat this, we use well-established settings [3], [11], [28] in our experiments, e.g., normalization, setting the granularity threshold to 10 LOC, and using a 0.3 dissimilarity threshold for Type-3 clones.

Internal validity. The manual classification of cloned functions and contracts (see Tables 2 and 4) can result in incorrectly classified data. Furthermore, because of the

broader definition of the categories, there is always room for interpretation when conducting the classification. To address this potential threat, we made the list of categorized functions available to public scrutiny.[10]

External validity. Our study has sampled only verified smart contracts deployed on the Ethereum platform, a subset of all smart contracts deployed on the platform. Thus, there are no guarantees on the safe generalization of our findings to all smart contracts written in Solidity. The same reasoning applies to generalizing our findings to other blockchain platforms. However, the goal of these experiments was not to provide a general theory for all Ethereum smart contracts, but to extract initial and high-level insights from existing smart contracts in order to raise awareness about the highly vulnerable state of systems relying on immutable code. We are still reasonably confident, that many of our insights translate well to other platforms relying on immutable source code. An external threat to validity w.r.t. the original study we could not mitigate is the number of transactions used in the analysis of RQ2, as explained in Section 6.2.2.

Limitations. Due to the limited parsing support for Solidity (especially compared to that for mainstream languages, such as Java and C++), we have developed a custom parser using the TXL grammar [30]. Since this is the first version of the parser, bugs and other shortcomings are possible. Although we have not experienced such issues during our experiments, we have made the parser available to public scrutiny[10]. Nevertheless, as a sign of maturity, the Solidity parsers and normalizers developed for our experiments have become part of NiCad starting with its v6.2 release.

10 Conclusion

In this paper, we reported the results of our study on source code cloning practices on the Ethereum blockchain platform. By analyzing 33,073 Solidity smart contracts, we found that 30.13% of the source code is cloned. Our work is an extended conceptual replication of the study of Kondo et al. [13] who reported a substantially higher clone ratio of 79.2%. The main difference between the two studies is the level of granularity clones are analyzed at. Our analysis was carried out at the level of functions, while the original analysis was carried out at the level of whole source files. To achieve this finer granularity of cloning analysis, we extended the NiCad clone detection tool to support Solidity, the programming language of the Ethereum platform. Our study reports a lower boundary of the clones on the Blockchain platform. This lower boundary is still on par with the 6–50% rate of cloning reported from traditional software engineering domains [2], [3], suggesting potential risks of reduced security, reliability, and performance of the overall software system.

An important takeaway of our study is that these problems could be effectively addressed by refactoring. The majority, about 90% of clones are of Type-1, i.e., exact replicas, and such clones have been shown to be easier to refactor [9]. Moreover, as shown by our cluster analysis, cloned functions tend to form hotspots in the source code: half of the clones can be found in just about 2% of clusters. Such clusters should be the prime candidates for refactoring. Unfortunately, the lack of coordination between developers renders such efforts particularly challenging. Thus, we anticipate automated audit mechanisms to appear in the integration (pre-deployment) phase of blockchain DevOps processes [51]. Solutions such as the NiCad-based tool presented in this paper could serve as a machinery to generate refactoring recommendations to reduce the clone ratio in the code to be deployed, thereby improving the overall code quality of the platform. Furthermore, we foresee the emergence of quality control as a service, provided by platform agents in exchange for compensation that is proportional to their computation investment.

Future work should focus on extending the scope of the current study to smart contract programming languages of other platforms, such as Script, the language of Bitcoin. Opportunities in adapting traditional software engineering lifecycle models to the particularities of smart contract development should be considered as well.

References

[1] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: platforms, applications, and design patterns," in *International conf. on financial cryptography and data security*. Springer, 2017, pp. 494–509.

[2] B. Laguë *et al.*, "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process," in *International Conference on Software Maintenance*. IEEE, 1997, pp. 314–321.

[3] W. Hasanain *et al.*, "An analysis of complex industrial test code using clone analysis," in *International Conference on Software Quality, Reliability and Security*. IEEE, 2018, pp. 482–489.

[4] C. J. Kapser and M. W. Godfrey, ""Cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[5] C. K. Roy *et al.*, "The vision of software clone management: Past, present, and future," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*. IEEE, 2014, pp. 18–33.

[6] R. Koschke, "Frontiers of software clone management," in *2008 Frontiers of Software Maintenance*. IEEE, 2008, pp. 119–128.

[7] D. Chatterji *et al.*, "Effects of cloned code on software maintainability: A replicated developer study," in *Working Conference on Reverse Engineering*. IEEE, 2013, pp. 112–121.

[8] E. Jürgens, F. Deissenboeck, and B. Hummel, "Code similarities beyond copy & paste," in *European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 78–87.

[9] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones," *IEEE Trans. Software Eng.*, vol. 41, no. 11, pp. 1055–1090, 2015.

[10] C. Roy and J. Cordy, "A survey on software clone detection research," Ontario, Canada, Tech. Rep. 2007-541, 2007.

[11] B. van Bladel and S. Demeyer, "Clone Detection in Test Code: An Empirical Evaluation," in *International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2020, pp. 492–500.

[12] C. Dannen, *Introducing Ethereum and solidity*. Springer, 2017.

[13] M. Kondo *et al.*, "Code cloning in smart contracts: a case study on verified contracts from the Ethereum blockchain platform," *Empirical Software Engineering*, vol. 25, no. 6, pp. 4617–4675, 2020.

[14] C. K. Roy and J. R. Cordy, "NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Int. Conf. on Program Comprehension*. IEEE, 2008, pp. 172–181.

[15] L. Jiang *et al.*, "DECKARD: scalable and accurate tree-based detection of code clones," in *International Conference on Software Engineering*. IEEE, 2007, pp. 96–105.

[16] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, 2009.

[17] R. K. Saha *et al.*, "Understanding the evolution of type-3 clones: an exploratory study," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE, 2013, pp. 139–148.