Noname manuscript No.
(will be inserted by the editor)

Code Cloning in Smart Contracts: A Case Study on Verified Contracts from the Ethereum Blockchain Platform

Masanari Kondo · Gustavo A. Oliva · Zhen Ming (Jack) Jiang · Ahmed E. Hassan · Osamu Mizuno

Received: date / Accepted: date

Abstract Ethereum is a blockchain platform that hosts and executes smart contracts. Smart contracts have been used to implement cryptocurrencies and crowdfunding initiatives (ICOs). A major concern in Ethereum is the security of smart contracts. Different from traditional software development, smart contracts are immutable once deployed. Hence, vulnerabilities and bugs in smart contracts can lead to catastrophic financial loses. In order to avoid taking the risk of writing buggy code, smart contract developers are encouraged to reuse pieces of code from reputable sources (e.g., OpenZeppelin). In this paper, we study code cloning in Ethereum. Our goal is to quantify the amount of clones in Ethereum (RQ1), understand key characteristics of clone clusters (RQ2), and determine whether smart contracts contain pieces of code that are identical to those published by OpenZeppelin (RQ3). We applied Deckard, a tree-based clone detector, to all Ethereum contracts for which the source code was available. We observe that developers frequently clone contracts. In particular, 79.2% of the studied contracts are clones and we note an upward trend in the number of cloned contracts per quarter. With regards to the characteristics of clone clusters, we observe that: (i) 9 out of the top-10 largest clone clusters are token managers, (ii) most of the activity of a cluster tends to be concentrated on a few contracts, and (iii) contracts in a cluster to be

✉ Masanari Kondo, Osamu Mizuno
Software Engineering Laboratory, Department of Information Science
Kyoto Institute of Technology, Kyoto, Japan
E-mail: m-kondo@se.is.kit.ac.jp, o-mizuno@kit.ac.jp

Gustavo A. Oliva, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL), School of Computing
Queen's University, Kingston, Canada
E-mail: {gustavo,ahmed}@cs.queensu.ca

Zhen Ming (Jack) Jiang
Department of Electrical Engineering & Computer Science
York University, Toronto, Canada
E-mail: zmjiang@cse.yorku.ca

created by several authors. Finally, we note that the studied contracts have different ratios of code blocks that are identical to those provided by the OpenZeppelin project. Due to the immutability of smart contracts, as well as the impossibility of reverting transactions once they are deemed final, we conclude that the aforementioned findings yield implications to the security, development, and usage of smart contracts.

Keywords Smart Contracts, Code cloning, Ethereum, Blockchain

1 Introduction

Ethereum is a blockchain platform (Wood, 2017). A blockchain platform is a distributed, chronological database of transactions that is shared and maintained across nodes that participate in a peer-to-peer network (Swan, 2015). The decentralized nature of a blockchain enables transactions to be processed without the need of a trusted third-party, such as a bank or a credit card company. Due to its unique properties, blockchain has attracted the attention of media outlets such as The Economist (2018) and The New York Times (2017). Industry-leading companies such as Facebook are also starting to develop their own blockchain platforms[1].

At the heart of the Ethereum platform are *smart contracts* (Szabo, 1994), which can be seen as general-purpose computer programs. Hence, Ethereum is often referred to as a programmable blockchain platform. The source code of a smart contract is typically written in Solidity, whose syntax resembles that of Java. The source code is organized in terms of *subcontracts* (similar to classes), *libraries* (similar to utility classes), and *interfaces* (identical to Java's interfaces). For convenience purposes, we indistinctly refer to subcontracts, libraries, and interfaces as *code blocks*.

A major concern in the Ethereum Platform is the security of smart contracts. The reason is twofold. First, different from traditional software development, the source code of smart contracts is *immutable*. Even if developers identify that their deployed smart contract has a security issue, they cannot simply apply a fix to the code. Second, prior research shows that the vast majority of smart contracts manage financial operations, in the sense that they operate on tokens that have a certain market capitalization. Therefore, the exploitation of bugs in a smart contract could result in large amounts of tokens (e.g., cryptocurrency) being stolen. In fact, such an unfortunate scenario has already occurred. A blockchain-based application known as "The DAO" launched with 150 million USD in crowdfunding in June 2016. Its smart contract was shortly-after hacked by exploiting a recursion call vulnerability. The attacker managed to drain 50 million USD worth of cryptocurrency.

More generally, since developers cannot change the source code of a smart contract after its deployment, we expect that developers will often reuse (and potentially clone) code from reputable sources instead of writing code from

[1] `https://libra.org`

scratch and taking the risk of introducing bugs. OpenZeppelin is a prominent example of a project devoted to creating secure libraries and template contracts to be reused by smart contract developers.

The goal of this paper is to quantify the amount of clones in Ethereum, understand key characteristics of these clone clusters (categories, activity level, and authorship), and determine whether code blocks developed by smart contract libraries (e.g, the OpenZeppelin project) are used in the studied contracts. We highlight that our study does not entail designing a new clone detector for smart contracts. Instead, we leverage an existing, robust tool (Deckard) that operates at the source code level in order to detect clones. More specifically, we investigate the following research questions.

RQ1) How frequently are verified contracts cloned? We downloaded the source code of all verified contracts from Etherscan. A verified contract is a contract for which the source code is made available on Etherscan (the primary dashboard website for Ethereum). Subsequently, we applied Deckard, a tree-based clone detector. We observed that:

Developers frequently clone contracts, as only 20.8% of the studied contracts are not a clone of any other contract. The percentage of clones among newly created contracts continues to increase over time. In the last quarter of the studied period (2018.Q2), almost 3/4 of all created contracts were clones. Finally, 43.3% of the studied contracts are type-2 clones (Bellon et al., 2007), suggesting that developers rely on templates to develop smart contracts.

RQ2) What are the characteristics of clusters of similar verified contracts? We analyzed the clone clusters generated by Deckard. Our goal was to derive factual insights regarding how these clusters are created and their characteristics. We observed that:

9 out of the top-10 largest clone clusters are token managers, most of the activity of a cluster tends to be concentrated on a few contracts, and contracts in a cluster to be created by several authors.

RQ3) How frequently code blocks of verified contracts are identical to those from OpenZeppelin? Given the reputation and goal of OpenZeppelin, we downloaded all OpenZeppelin releases, extracted their code blocks, and determined if code blocks from verified contracts are identical to code blocks from OpenZeppelin. We observed that:

The studied contracts have different ratios of code blocks that are identical to those provided by the OpenZeppelin project. The ERC20 OpenZeppelin category is the most frequently reused category, which contains code blocks to support the implementation of token contracts that comply with the ERC20 standard.

The main take-away of this study is that code cloning is a common practice in Ethereum. Due to the architectural characteristics of Ethereum (e.g., immutability of smart contracts and the impossibility to revert transactions), the prevalence of code clones yield practical implications to the security, devel-

opment, and usage of smart contracts. The data produced as part of this study is made available online in the form of a *supplementary material* package[2].

This remainder of this paper is organized as follows. In Section 2, we briefly define key concepts surrounding blockchains and smart contracts. Since not all reader might be familiar with the intricacies of Ethereum, we provide a more in-depth background in Appendix A. In Section 3, we explain the data collection procedures that we applied to obtain the source code and the associated metadata of smart contracts. In Section 4, we present a preliminary study to motivate this paper. In Section 5, we describe how we choose a clone detector and set up its parameters. In Section 6, we present the motivation, approach, and results for each of the research questions that we address in this paper. In Section 7, we discuss the practical implications of our findings. In Section 8, we contextualize our research in the code cloning domain, as well as present different perspectives from which prior research has studied smart contracts. In Section 9, we discuss the threats to the validity of our study. Finally in Section 10, we state our conclusions.

2 Background

In this section, we summarize the concepts that are key to this study. Since not all readers might be familiar with the intricacies of Ethereum, we included an Appendix A that provides a more thoroughly explanation of these concepts.

2.1 Ethereum

Blockchain. A blockchain is a distributed, chronological database of transactions that is shared and maintained across nodes that participate in a peer-to-peer network.

Ethereum. As opposed to Bitcoin, Ethereum is a blockchain platform that supports *smart contracts*. More specifically, Ethereum both hosts and executes smart contracts. Because of these attributes, Ethereum is often referred to as a *programmable blockchain*.

Smart contract. A smart contract is a *general-purpose computer program*. In Ethereum, smart contracts are typically written in the Solidity programming language. Solidity is object-oriented and its syntax resembles that of Java. The source code of a Solidity smart contract is organized in terms of *subcontracts*, *interfaces*, and *libraries*. We indistinctly refer to these three constructs as *code blocks*.

Verified smart contract. Ethereum only stores the bytecode of smart contracts (i.e., the source code is not available). A *verified* smart contract is a smart contract that has a flattened version of its source code published on

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

Etherscan (a popular Ethereum explorer website). We refer to this flattened version of the source code as the *code file* of a verified contract.

Ethereum accounts. Ethereum has two types of accounts: user accounts and smart contract accounts. Both types of accounts have a unique ID. User accounts can *deploy* smart contracts. The deployer is commonly referred to as the *creator* (or *author*) of the contract. The deployment of a contract is analogous to an object instantiation (as in object-oriented programming): an instance (object) is created out of the smart contract (class) and stored in the blockchain. Once deployed, the code of a smart contract cannot be changed or replaced. User accounts can also *execute* deployed smart contracts by sending transactions to invoke functions defined in these contracts.

Cryptocurrency, tokens, and coins. A *cryptocurrency* is digital and represents money. A cryptocurrency is native to its own blockchain. In the case of Ethereum, the cryptocurrency is called Ether and is abbreviated as ETH. Ether can be transferred between user accounts. *Tokens* are created on top of existing blockchains. Tokens are used to represent digital assets that are tradeable. Every token has a name and an acronym (popularly known as a *symbol*) and any smart contract can define a new token. It is common for tokens to represent money. Therefore, in practice, coins and tokens are frequently used interchangeably.

Token contract. A token contract is a smart contract that defines a token and keeps track of its balance.

2.2 Code Cloning

Clones are pieces of code that are either identical or "very similar". In order to differentiate between the different shades of cloning, Bellon et al. (2007) proposed a simple code cloning classification:

Type-1 Clone. Identical code fragments except for variations in whitespace, layout and comments.

Type-2 Clone. Syntactically identical fragments except for variations in identifiers, literals, whitespace, layout and comments.

Type-3 Clone. Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, whitespace, layout and comments.

We study type-1 and type-2 clones as part of RQ1 (Section 6.1). Since the detection of type-3 clones is still an active research area (Sajnani et al., 2016), its detection is out of the scope of our study.

3 Data Collection

All the data that are used in this study were collected from the Etherscan website[3] and the OpenZeppelin repository[4]. Etherscan is the primary dashboard for Ethereum. Etherscan operates by having several nodes in the Ethereum blockchain network, which capture the state of the network (e.g., transactions being sent and blocks being created). The gathered data is then published on the website, allowing people to explore Ethereum though a web browser. Etherscan also offers a REST API, however it does not cover all published data in the website. OpenZeppelin is one of the most popular repositories of building blocks for developing secure smart contracts. We discuss the relevance of OpenZeppelin in more details as part of RQ3 (Section 6.3).

In the following, we describe the specific pieces of data that we collected in order to answer our research questions. An overview of our data collection process is shown in Figure 1.

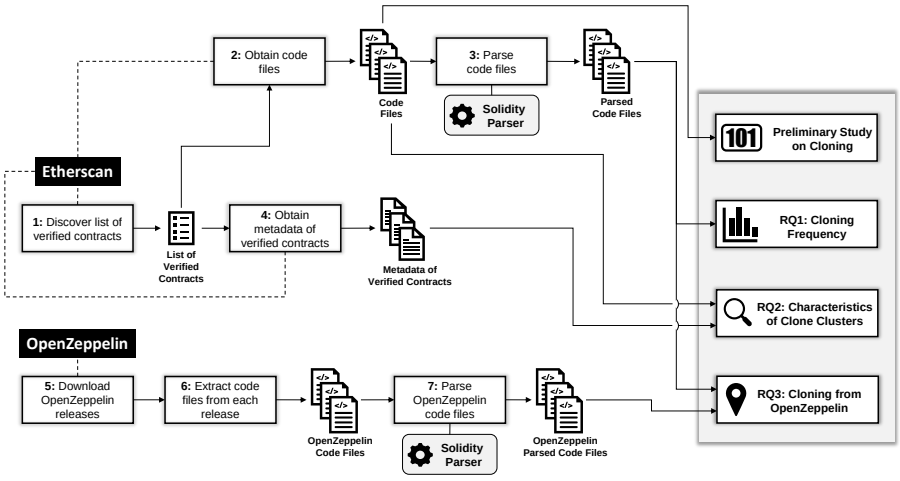

Fig. 1: An overview of our data collection process. Dashed lines indicate a connection to a data source.

1: Discover list of verified contracts. The list of verified contracts available on Etherscan represents the starting point of our data collection process. Such a list is not recoverable via the API. However, it is available on the Etherscan website in the form of a paginated table[5]. Therefore, we recorded all entries of this table. At the time of our data collection, there were 33,073 verified contracts on Etherscan.

[3] `https://etherscan.io`

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

[5] `https://etherscan.io/contractsVerified`

2: Obtain code files. At the time of our data collection, there was no method available in the API to download the Solidity code files. Therefore, we obtained the code files directly from the web interface. These code files are used as input to our preliminary study.

3: Parse code files. To extract the abstract syntax tree (AST) of code files, we used the Solidity parser made available by Federico Bond[6]. The parsed code files were used as input to RQ1 (to support the detection of type-2 clones) and RQ3 (to enable the identification of code blocks within a code file).

4. Obtain metadata of verified contracts. Since cryptocurrencies play a key role in Ethereum, we wanted to differentiate between token code and non-token contracts (Section A.4.1). Therefore, we went through the Etherscan webpage of each verified contract, searching for the "Token Tracker" field. This field shows the name and symbol of the token that is associated with a contract. The field is absent when the contract does not define or manage a token. We also retrieved the number of received transactions and the creator of each contract. These metadata support the investigations conducted in RQ2.

5: Download OpenZepplin releases. In order to determine the identical piece of codes between the contracts, libraries and interfaces (code blocks) from Ethereum and those from OpenZeppelin, we downloaded all OpenZeppelin releases available in the project's GitHub repository (Table 1).

Table 1: Studied OpenZeppelin releases.

Version	Number of Code Files	Release Date (Latest Commit)	LoC
v1-0-0	36	2016-11-24	1,112
v1-1-0	37	2017-07-02	2,327
v1-2-0	38	2017-07-18	2,341
v1-3-0	40	2017-09-21	2,273
v1-4-0	42	2017-11-23	2,199
v1-5-0	66	2017-12-22	2,488
v1-6-0	76	2018-01-23	3,144
v1-7-0	90	2018-02-20	3,722
v1-8-0	100	2018-03-23	4,218
v1-9-0	106	2018-04-26	4,560
v1-10-0	106	2018-06-05	4,976
v1-11-0	112	2018-07-13	5,090
v1-12-0	115	2018-08-10	5,182

6: Extract code files from each release. Each published OpenZeppelin release consists of a zip file that contains a set of code files. Therefore, in this step we simply decompress the zip files.

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

Summary of Data Collection

- Data collection starting day: 2018-07-03 (yyyy-mm-dd).
- Data sources: Etherscan and OpenZeppelin (GitHub repo).
- Collected data: Code file and associated metadata (token information, number of transactions, author) from verified contracts. Code files from OpenZeppelin releases published in GitHub.
- Number of verified code files: 33,073.
- Number of OpenZeppelin releases: 13.

4 Preliminary Study

Motivation. While manually browsing through the source code of verified contracts on Etherscan, we noticed that they generally seemed to be very similar to one another. Such a subjective observation motivates this preliminary study. Our goal is to objectively quantify how similar verified contracts are to one another.

Approach. We evaluate similarity in terms of *pair-wise textual similarity*. As a result of our data collection (Section 3), we obtained the code file of each of the 33,073 verified contracts. Due to the large amount of code files, we refrained from using common edit distance algorithms (e.g., Levenshtein distance, Longest Common Subsequence, Hamming distance, and Jaro–Winkler distance) because of their complexity, which is $\mathcal{O}(m \times n)$ for a pair of strings with lengths m and n. Instead, we rely on Q-Grams (Ukkonen, 1992), which can be computed in $\mathcal{O}(m + n)$ time. A Q-Gram is a subsequence of q *consecutive* characters of a string. The *Q-Gram profile* of a string is a vector with the counts of all q-grams in that string.

Consider the example shown below, which depicts the Q-Gram profiles of strings $x = abcab$ and $y = cdabg$ for $q = 2$. For instance, the Q-Gram profile of x is the vector $\langle x_i, x_{i+1}, ..., x_n \rangle = \langle 2, 1, 1, 0, 0, 0 \rangle$, where n is the number of distinct Q-Grams found over all strings being compared. From this representation, several vector-based distance measures can be calculated. We employ the *Q-Gram distance* algorithm, which is given by the sum over the absolute differences $|x_i - y_i|$. For our running example, the Q-Gram distance is $|2-1| + |1-0| + |1-0| + |0-1| + |0-1| + |0-1| = 6$.

From the Q-Gram distance, we finally compute the *Q-Gram similarity*. Q-Gram similarity is computed by dividing the Q-Gram distance by the maximum possible distance and then subtracting the result from 1. This calculation results in a score between 0 and 1, with 1 corresponding to complete similarity and 0 to complete dissimilarity. In the discussed example, the maximum possible difference is 8, since the length of the Q-gram profiles is 4 and we have two strings $(4 * 2)$. Therefore, the Q-Gram similarity between the strings x and y is $1 - (6/8) = 1 - (3/4) = 0.25 = 25\%$.

```
               ab  bc  ca  cd  da  bg
x = "abcab"    2   1   1   0   0   0
y = "cdabg"    1   0   0   1   1   1
```

We compute the Q-Gram similarity for all pairs of code files. We pick $q = 4$, which has been shown to be adequate for code completion tasks (Roos, 2015) and in the analysis of textual repetition in source code (Hindle et al., 2012). We say that two contracts are *very similar* if the Q-Gram similarity of their code files is higher than or equal to 90%. In terms of implementation, we use the `stringsim` function from the `stringdist` R package.

Result. ***81.9% of the verified contracts are very similar to at least one other verified contract.*** Figure 2 depicts the similarity between all contract pairs. Only 0.92% of all contract pairs have a Q-Gram similarity of at least 90%.

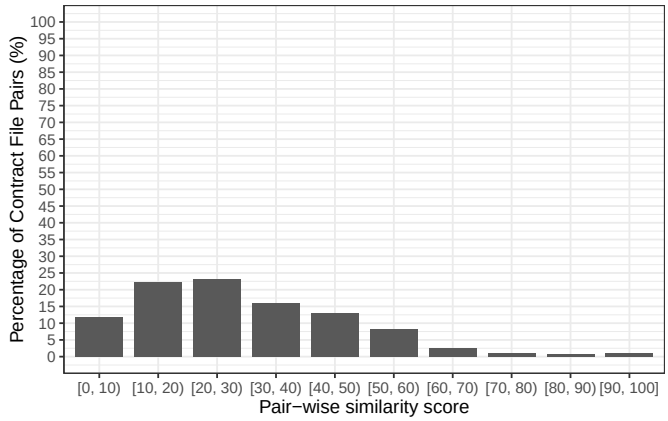

Fig. 2: Histogram showing the percentage of contract pairs that are similar at a certain Q-Gram similarity interval.

However, one should note that the number of contract pairs is remarkably large (544,483,500). If we focus on contracts *individually* in lieu of pairs, we observe that 81.9% of the contracts are very similar to at least one other contract (Figure 3). Moreover, 50.2% of the verified contracts are very similar to at least 8 other contracts. Interestingly, the long tail of the curve indicates that some specific verified contracts are very similar to a noteworthy number of other contracts. For instance, 15.4% of the contracts are very similar to more than 1,000 other contracts. The extreme case reveals a contract that is very similar to 2,561 other contracts.

In summary, the vast majority of verified contracts share high textual similarity with at least one other verified contract. This finding drives the remainder of this paper.

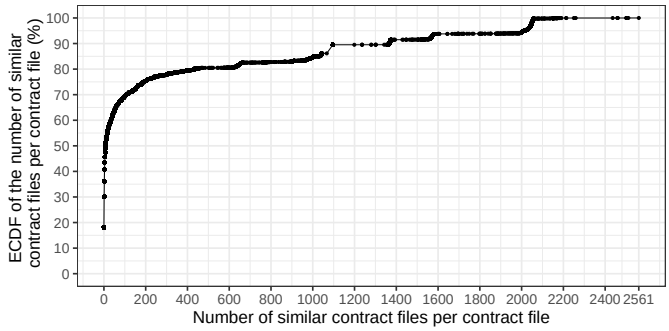

Fig. 3: The Empirical Cumulative Distribution (ECDF) for the number of similar contracts.

5 Experimental Setup

In the preliminary study (Section 4), we observed that the vast majority of verified contracts are very similar to at least one other contract. Such a result encouraged us to further study code cloning in Ethereum with a proper clone detector. In this section, we describe the clone detector that we selected and how we set up its parameters.

5.1 Selection of a clone detector

Since the focus of this study is not to develop our own detector nor benchmark existing detectors, we favoured the selection of a mature, scalable, and Solidity-compatible clone detector that would be ready-to-use out of the box. We selected Deckard (Jiang et al., 2007a,b), which is a fast, tree-based clone detector. Support for Solidity was added on v2.0, released on May 25th, 2018[7].

The architecture of Deckard is shown in Figure 4. Jiang et al. (2007a) describe it as follows: *(1) A parser is automatically generated from a formal syntax grammar; (2) The parser translates sources files into parse trees; (3) The parse trees are processed to produce a set of vectors of fixed-dimension, capturing the syntactic information of parse trees; (4) The vectors are clustered w.r.t. their Euclidean distances; and (5) Additional postprocessing heuristics are used to generate clone reports.*

Deckard takes as input a list of smart contracts (Solidity files) and outputs a list of *clone clusters*. Each clone cluster contains a group of contracts that are clones of each other. Each contract resides in a single

[7] `https://github.com/skyhover/Deckard`

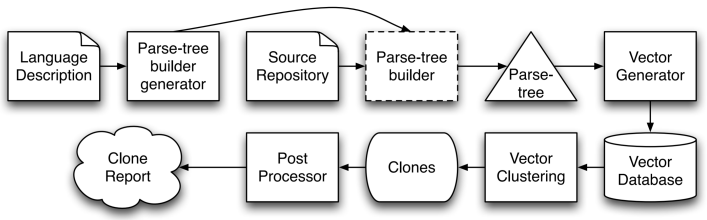

Fig. 4: Deckard's architecture (Jiang et al., 2007a).

5.2 Clone detection parameters

Deckard has three parameters, namely: `min_tokens`, `stride`, and `similarity`. The `min_tokens` is a minimum token count intended to suppress characteristic vectors for small subtrees. In other words, it helps to avoid reporting too small clones that are often uninteresting. The `stride` parameter determines how far a sliding window moves in each step during vector generation. According to the authors, larger strides reduce the amount of overlapping among tree fragments and may thus reduce the number of spurious clones. Finally, `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.

In their study, Jiang et al. (2007a) evaluate Deckard using several parameter values: `min_tokens` = {30,50}, `stride` = {2,4,8,16,Inf.}, and `similarity` = {0.90, 0.95, 0.99, 0.999, 0.9999, 1.0}. In a manual inspection of the results obtained by running Deckard on JDK 1.4.2 with `min_tokens` = 50, `stride` = 4, and `similarity` = 1.0, the authors observed that 93% of the clones reported by Deckard were actual clones. We employ the notation $c = <50,4,1.0>$ to succinctly refer to this parameter configuration.

5.3 Sensitivity Analysis

Given the aforementioned result on JDK, we decided to use parameter configuration $c = <50,4,1.0>$ as a starting point for our sensitivity analysis. Our goal is to find a parameter configuration that is suitable for our study of code clones on Ethereum. Given the results that we obtained in our preliminary study, *we focus on the detection of clones among code files* (in lieu of code fragments or subcontracts).

Deckard detects clones between code fragments. Since `min_tokens`=50 and `stride`=4 were presumably large enough for the detection of cloned code fragments, we also choose these values for the detection of cloned Solidity code files. Therefore, we focus our sensitivity analysis on the `similarity` threshold. As described by Jiang et al. (2007a), the similarity score needs to be very

high in order to achieve good precision for small cloned fragments. However, since we are focusing on code files, it might be possible that lower similarity thresholds need to be employed. Therefore, we start with a list of known clone pairs (from practical experience) and set the similarity threshold to 1.0. We then evaluate how many of the known clone pairs Deckard is able to detect and repeat the process using a lower `similarity` threshold until a good balance is achieved. Table 2 depicts the clone pairs that we use for our sensitivity analysis.

Table 2: Known clone pairs employed to support the choice of a suitable similarity threshold for Deckard.

Clone Pair	Contract 1	Contract 2
1	0x613d0e9b91af3d0057fe376194580dd0048e91d4	0x0016e71c7ced04b51a1fd8bb5c36d9e0cee9e1bb
2	0x00b9034425e357bf61b4abeb22299ec4a62c725b	0x001b6e5c7322899355eb65486e8cbb7dbbf19127
3	0x045c60df00aa2b7ff1753ee81b57bfe5e290e732	0x066b2063d4bd8cade177c55de659884c40bf2b8f
4	0x40e9a10fecd0350305d5e919e3963248fd6f845d	0x003516572f212dec785c3bcc97e6f354851eeb47
5	0x0035743c08768ad6558b49d751e0215762057754	0x008f81cbd97a3f59291aa0fed45a42491f10cfd2
6	0x23221fe28dadf788c7c59d0367bafef3b1607344	0x004460229a42542772f21ee82b8772cc6f2a502b
7	0x007b749fd9c28455f03a57c005f4249693550e51	0x015a8a0163ad54c86012ff57d3558b6271a2d2bd
8	0x00cf36853aa4024fb5bf5cc377dfd85844b411a0	0x00674045bb7c17f0aa1cde34780d6c51af548728
9	0x00eb84e7ca4ad6dbbef240056bc904003029a4cd	0x1b5d56bfe749e492ae226cf9aa23c1426f828b7b
10	0x09f55c2d116a5833d41ba9208216d11a7cdba4b3	0x0160ab3faf146f346b2cea49a7049d786aa1aafb

The results of our sensitivity analysis are summarized in Table 3. Using `similarity` = 1.00, we only detected 3/10 clone pairs. We reduced the `similarity` threshold in 0.05 decrements and the number of detected clone pairs increased. When we used `similarity` = 0.75, we were able to identify 8/10 clone pairs. From that point, we could only find an additional clone pair by bringing the `similarity` threshold all the way down to 0.40. We thus concluded that a reasonable similarity threshold lied in the [0.75, 0.80] interval. Hence, starting from 0.80, we reduced the *similarity* in 0.01 decrements. With `similarity` = 0.79, we were able to detect 8/10 clone pairs. This was our final `similarity` threshold choice.

Summary of experimental setup

- We use Deckard to detect clones between Solidity code files due to its reliability, scalability, and out-of-the-box support for Solidity.
- By means of a sensitivity analysis conducted with 10 known clone pairs (a priori), we achieved the following Deckard parameter configuration: `min_tokens`=50, `stride`=4, and `similarity`=0.79.

6 Empirical Study on Code Cloning

In this section, we show our three research questions, the approach that we employed in order to answer them, and our findings. In RQ1, we analyze the

Table 3: Sensitivity analysis: determining a suitable similarity threshold

	Clone Group										
Similarity	1	2	3	4	5	6	7	8	9	10	**Detected**
1.00					✓	✓	✓				3/10
0.95	✓				✓	✓	✓		✓		5/10
0.90	✓			✓	✓	✓	✓		✓		6/10
0.85	✓			✓	✓	✓	✓		✓		6/10
0.80	✓			✓	✓	✓	✓		✓	✓	7/10
0.79	✓	✓		✓	✓	✓	✓		✓	✓	8/10
0.75	✓	✓		✓	✓	✓	✓		✓	✓	8/10
0.70	✓	✓		✓	✓	✓	✓		✓	✓	8/10
0.65	✓	✓		✓	✓	✓	✓		✓	✓	8/10
0.60	✓	✓		✓	✓	✓	✓		✓	✓	8/10
0.55	✓	✓		✓	✓	✓	✓		✓	✓	8/10
0.50	✓	✓		✓	✓	✓	✓		✓	✓	8/10
0.45	✓	✓		✓	✓	✓	✓		✓	✓	8/10
0.40	✓	✓	✓	✓	✓	✓	✓		✓	✓	9/10

prevalence of clones (Section 6.1). In RQ2, we describe key characteristics of clone clusters (Section 6.2), namely: category, activity, and authorship. Finally, in RQ3, we determine whether the studied contracts have subcontracts that are clones of the contracts provided by the OpenZeppelin project (Section 6.3).

6.1 RQ1: How frequently are verified contracts cloned?

Motivation. In the preliminary study, we observed that the vast majority of verified contracts are very similar to at least one other contract. In this research question, we use a robust clone detector in order to find clones among code files in Ethereum.

The key motivation behind detecting clones in Ethereum is that *deployed smart contracts are immutable*. As opposed to traditional software development, developers cannot change the code if a bug is discovered. Moreover, smart contracts frequently hold a balance in cryptocurrency (Ether) and tokens (which have a market value). Hence, the security of smart contracts is a key concern in the Ethereum domain. Now let us assume that there exists a group of 100 contracts with identical source code. If a hacker manages to discover an exploitable vulnerability in the source code, then 100 contracts are instantly at risk. Discovering the prevalence of cloning and the size of clone clusters will provide insights into this matter.

Approach. In this RQ, we analyze the output of Deckard. We first report the clone ratio, i.e., the percentage of contracts that are flagged as a clone of some other contract. Subsequently, we sort clusters in descending order of size and plot the percentage of clusters against the cumulative percentage of contracts. Next, we perform a historical analysis to identify trends. More specifically, we check how many clones have been created in each quarter of the analysis period.

Following this overview, we leverage the clone classification of Bellon et al. (2007) (Section 2) and detect type-1 and type-2 clones. These two types of clones are detected by *post-processing* the output of Deckard. Figure 5 summarizes our approach. In the following, we describe the steps in detail.

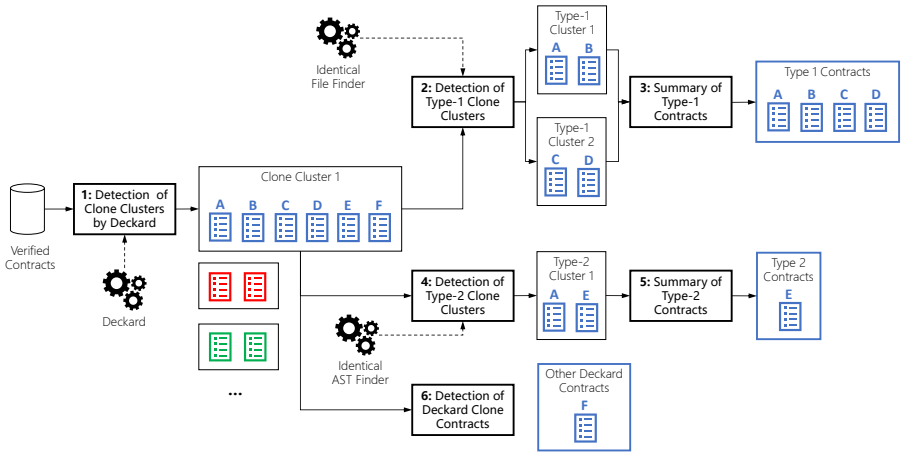

Fig. 5: Summary of our clone detection approach.

1. Detection of Clone Clusters by Deckard. We detect clone clusters using the approach described in Section 5. The detection of type-1 and type-2 clones is then performed for each clone cluster. In Figure 5, we use Clone Cluster 1 as an example.

2. Detection of Type-1 Clone Clusters. We strip whitespaces, tabulations (tabs), and comments from the code file of each verified contract. After this preprocessing stage, we compare code files in a pair-wise fashion. If two code files are identical, then they are put in the same type-1 cluster. *Type-1 Cluster 1* and *Type-1 Cluster 2* represent clusters of type-1 contracts.

3. Summary of Type-1 Contracts. To identify type-1 contracts, we simply take the union of all type-1 clusters. In the example shown in Figure 5, contracts A, B, C, and D are classified as type-1 contracts.

4. Detection of Type-2 Clones. We compute the abstract syntax tree (AST) of the code file of each contract and then search for identical ASTs. The only allowed exception is leaf nodes, which represent identifiers, literals, and other terminal constructs (e.g., compiler version, function modifiers). If two code files have identical ASTs, then they are added to the same cluster. In order to extract the ASTs, we used the Solidity parser that was developed by Federico Bond. Such parser works on top of a robust ANTLR4 grammar for Solidity, which was also developed by Bond[8]. Interestingly, Deckard relies on this

[8] `https://github.com/solidityj/solidity-antlr4`

```
sourceUnit
  : (pragmaDirective | importDirective | contractDefinition)* EOF ;

pragmaDirective
  : 'pragma' pragmaName pragmaValue ';' ;

contractDefinition
  : ( 'contract' | 'interface' | 'library' ) identifier
  ( 'is' inheritanceSpecifier (',' inheritanceSpecifier )* )?
  '{' contractPart* '}' ;

contractPart
  : stateVariableDeclaration
  | usingForDeclaration
  | structDefinition
  | constructorDefinition
  | modifierDefinition
  | functionDefinition
  | eventDefinition
  | enumDefinition ;

functionDefinition
  : 'function' identifier?
    parameterList modifierList returnParameters?
    ( ';' | block ) ;

parameterList
  : '(' ( parameter (',' parameter)* )? ')' ;

parameter
  : typeName storageLocation? identifier? ;

modifierList
  : ( modifierInvocation | stateMutability | ExternalKeyword
  | PublicKeyword | InternalKeyword | PrivateKeyword )* ;

returnParameters
: 'returns' parameterList ;

numberLiteral
  : (DecimalNumber | HexNumber) NumberUnit? ;

StringLiteral
: '"' DoubleQuotedStringCharacter* '"'
| '\'' SingleQuotedStringCharacter* '\'' ;
```

Listing 1: Excerpt of the ANTLR4 grammar for Solidity used in this study. The complete grammar can be found at `https://github.com/solidityj/solidity-antlr4/blob/master/Solidity.g4`.

same grammar to detect clones. For illustrative purposes, Listing 1 depicts an excerpt of the grammar.

Finally, we prune type-2 clusters to ensure that every two contracts in a cluster are different (i.e., not a type-1 pair). The rationale is to avoid inflating type-2 clusters due to the existence of type-1 pairs. Assume E is a type-2 clone of A. Since A and B are identical, E is also a type-2 of B. However, due the aforementioned pruning, we only add A to the cluster (Figure 5).

5. Summary of Type-2 Clones. Type-2 contracts are detected by taking the union of Type-2 clusters and then removing those contracts that had already been classified as type-1. In the example shown in Figure 5, only E is classified as a type-2 contract (since A had already been classified as a type-1 contract).

6. Detection of Deckard Clone Contracts. All contracts not classified as type-1, nor type-2, are classified as Deckard clones.

After detecting type-1 and type-2 contracts, we conduct exploratory analyses. In particular, we compare the length of the code file of type-1 clones

to that of other contracts and also investigate the kinds of AST nodes that are frequently modified in type-2 clone pairs (i.e., every two contracts inside a type-2 cluster). To conclude the section, we comment on the most frequently cloned contract per cloning type.

We note that, when convenient, we perform statistical tests and compute the Cliff's Delta (δ) effect size score. The effect size score helps us to understand the practical significance of the difference between two distributions. We assess Cliff's Delta using the following thresholds (Romano et al., 2006): *negligible* for $|\delta| \leq 0.147$, *small* for $0.147 < |\delta| \leq 0.33$, *medium* for $0.33 < |\delta| \leq 0.474$, and *large* otherwise.

Findings. Observation 1) ***79.2% of the verified contracts are clones.*** In other words, only 20.8% of the verified contracts are not a clone of any other verified contract.

Observation 2) ***There is a large number of clone clusters. Nevertheless, a small portion of clusters encompasses the vast majority of contracts.*** Applying Deckard resulted in 10,786 clusters. Figure 6 illustrates the relationship between the proportions of clusters and contracts. The shape of the curve indicates that a small portion of clusters encompass the majority of the contracts. Just as a reference, 20% of the clusters encompass approximately 68% of the contracts. In other words, there are a few remarkably large clusters of very similar contracts.

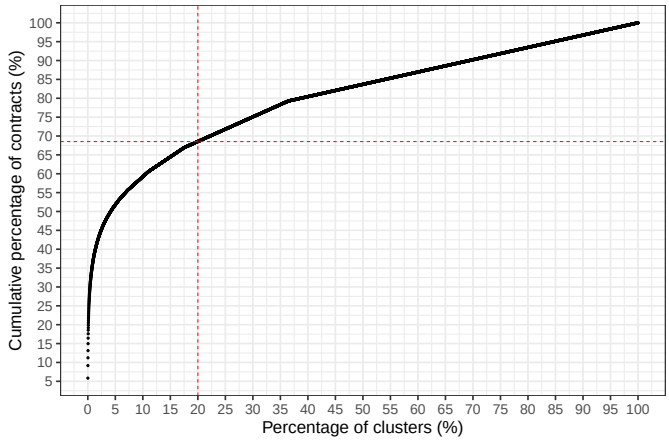

Fig. 6: Relationship between the proportions of clusters and contracts. A small portion of the clusters (20%) encompass the majority of the contracts (68%).

Observation 3) ***The percentage of clones among newly created contracts continues to increase over time.*** Figure 7 shows the percentage of clones among newly created contracts for every quarter. As indicated by the

purple bar, the total percentage of clones continues to increase over time. In particular, almost 75% of the contracts created in the last quarter (2018.2) are clones of preexisting contracts.

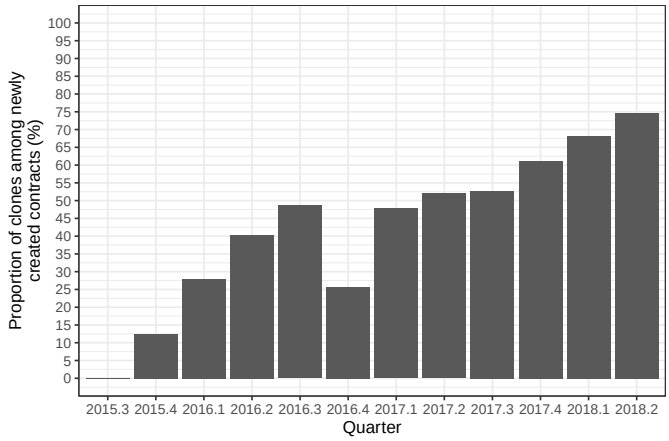

Fig. 7: Evolution of the percentage of cloned contracts among newly created contracts for every quarter.

Observation 4) ***Of all studied contracts, 16.7% are type-1 clones. Type-1 clones have lengthier code files compared to other contracts.*** One might expect developers to only *entirely* clone smaller contracts with a few functions. However, our data shows the opposite scenario. As depicted in Figure 8, the code files of type-1 clones have a higher number of instructions (semicolons plus open curly brackets) and code blocks (subcontracts, libraries, and interfaces) compared to the code files of other contracts. A two-sided non-paired Mann-Whitney test indicates that the difference is statistically significant at $\alpha = 0.05$ for both variables (p-value $<$ 2.2e-16 in the two cases). We observe that the effect sizes are $\delta = 0.333$ (medium) for the number of instructions and $\delta = 0.297$ (small, though non-negligible) for the number of code blocks.

Observation 5) ***Of all studied contracts, 43.3% are type-2 clones. Type-2 clone pairs commonly have almost identical code files. The few differences tend to be in literals, contract definitions, and function definitions.*** Type-2 clones are the most frequently occurring type of code clone in our dataset (16.7% are type-1 clones and 19.2% are other Deckard-identified clones). In order to better understand how developers perform type-2 cloning, we count the number of modified leaf nodes per type-2 clone pair and calculate the proportion over all leaf nodes. As indicated in Figure 9, the proportion of changed leaf nodes is very small for the vast majority of type-2 clone pairs. More specifically, this proportion is at most 5%

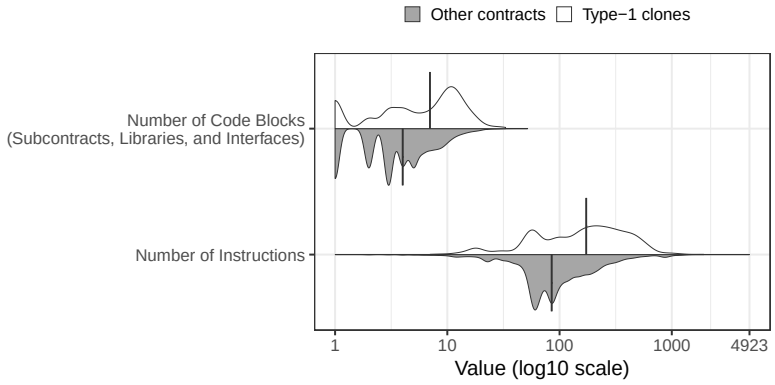

Fig. 8: Contracts classified as type-1 clones have a higher number of code instructions and code blocks compared to all other studied contracts (clones and non-clones). The x-axis indicates the value for each metric (i.e., number of code blocks and number of instructions). The y-axis indicates the metric.

for 99.7% of the type-2 clone pairs. In absolute terms, the number of changed leaf nodes is at most 10 for 98.3% of the type-2 clone pairs.

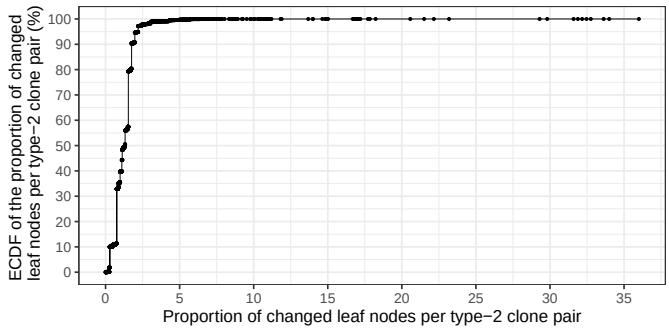

Fig. 9: ECDF of the proportion of changed leaf nodes per type-2 clone pair.

We also investigate the kinds of leaf nodes that are modified in type-2 clone pairs. Figure 10 depicts the proportions of each kind over all occurrences of leaf node modifications. `NumberLiteral`, `StringLiteral`, `FunctionDefinition`, and `ContractDefinition` account together for 95.1% of all leaf node modifications. With the exception of `PragmaDirective`, all other AST nodes represent (individually) less than 1% of all leaf node modifications.

Figure 11 shows an example of a type-2 clone pair in which leaf nodes of kinds `NumberLiteral`, `StringLiteral`, `FunctionDefinition`, and `ContractDefinition` were modified. The two contracts are *token contracts*

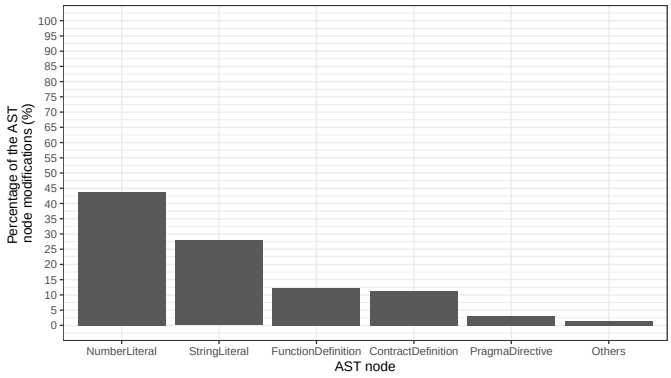

Fig. 10: Histogram showing the proportions of modifications per type of AST leaf node

(Section A.4.1). The changed `NumberLiterals` correspond to the coin supply (`_totalSupply`), the coin decimals (`decimals`), and account addresses (`balances` map). The coin supply indicates the maximum supply of a coin. The decimals indicate the minimum unit of a coin. For example, if decimals is 3, then the minimum coin unit is 0.001. The account addresses point to either a user account or a smart contract account (Section A.2). In the example (Figure 11), the balance of a certain hardcoded account (likely the contract creator account) is being updated with all the available coin supply. The changed `StringLiteral` correspond to the coin's name (`name`) and the coin's symbol (`symbol`). The changed `FunctionDefinition` corresponds to the function's name (in this case, the constructor). Finally, the changed `ContractDefinition` correspond to the subcontract's name.

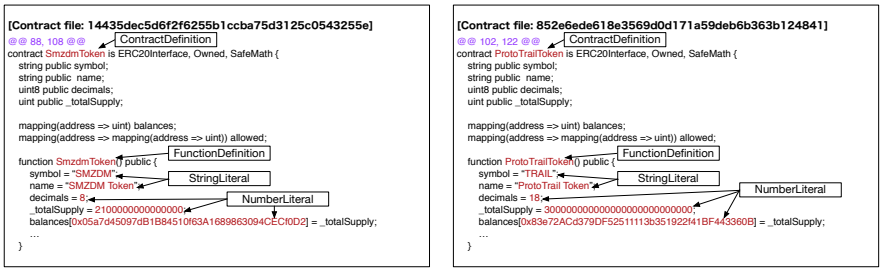

Fig. 11: An example of a type-2 clone pair that has the `NumberLiteral`, `StringLiteral`, `FunctionDefinition`, and `ContractDefinition` nodes with different values. The strings in red highlight nodes of the same type, but with different values. The strings in purple indicate the start and end lines of the shown code chunks. We removed blank lines for aesthetic reasons.

6.1.1 Observations on the contracts with the highest number of clone siblings

In this section, we investigate the contract with the highest number of clone siblings per cloning-type category. After detecting the contract with the highest number of clone siblings per cloning-type category, we manually analyze its code file in order to understand the key features that it implements. Our goal is to gain some insight into the contracts that are most frequently cloned by smart contract developers.

6.1.1.1 On the contract with the highest number of type-1 clone siblings

Since type-1 cloning is transitive (if contract C_1 is identical to C_2 and C_2 is identical to C_3, then C_1 is identical to C_3), it suffices to describe any contract in the largest group of type-1 clones. Our analysis revealed that the largest group of type-1 clones contains 169 mutually identical contracts. To serve as a reference, the address of one of the contracts in this group is `0x831979b878dd27f703a19fd1dcaddc2d0b425ac8`. Its code file is surprisingly complex, with 480 instructions (5.45 times higher than the median for code files outside this group) and 13 code blocks (3.25 times higher than the median for code files outside this group). The UML diagram (drawn by us) shown in Figure 12 depicts the code blocks of this contract, as well as their interrelationships.

The main goal of this contract is to implement flexible crowdsales with mintable tokens (Section A.4.2).

The code blocks in the *ERC Token Implementation* group (blue shaded box) define an abstract implementation of a token that follows the ERC20 standard (Section A.4.1). The `SafeMath` subcontract implements simple mathematical operations in a robust way by accounting for overflows, underflows, and division by zero.

The code blocks in the *Flexible Crowdsale Implementation* are the core pieces of this contract. The `CrowdsaleExt` subcontract implements a flexible crowdsale by supporting several features, including funding goals, refunds, and statistics for the crowdfunding. We emphasize that this subcontract defines several function modifiers (guards) that put preemptive restrictions on the execution of functions. For instance, the `«onlyOwner»` modifier guarantees that only the contract owner can execute the function (e.g., the `finalize()` function). Interestingly, this subcontract also defines two run-time extension points (hotspots), namely: `PricingStrategy` and `FinalizeAgent`. The `PricingStrategy` defines the pricing strategy, which may include presales, different tiers, etc. The `FinalizeAgent` defines what should happen once the crowdsale is completed. Anyone who deploys this contract can invoke the `setPricingStrategy(addr)` and the `setFinalizeAgent(addr)` to define the concrete implementations that should be used during runtime. However, the CrowdsaleExt is still an abstract contract, meaning that it has functions without implementation. The `MintedTokenCappedCrowdsaleExt` subcontract provides an implementation for the `CrowdsaleExt`, which uses

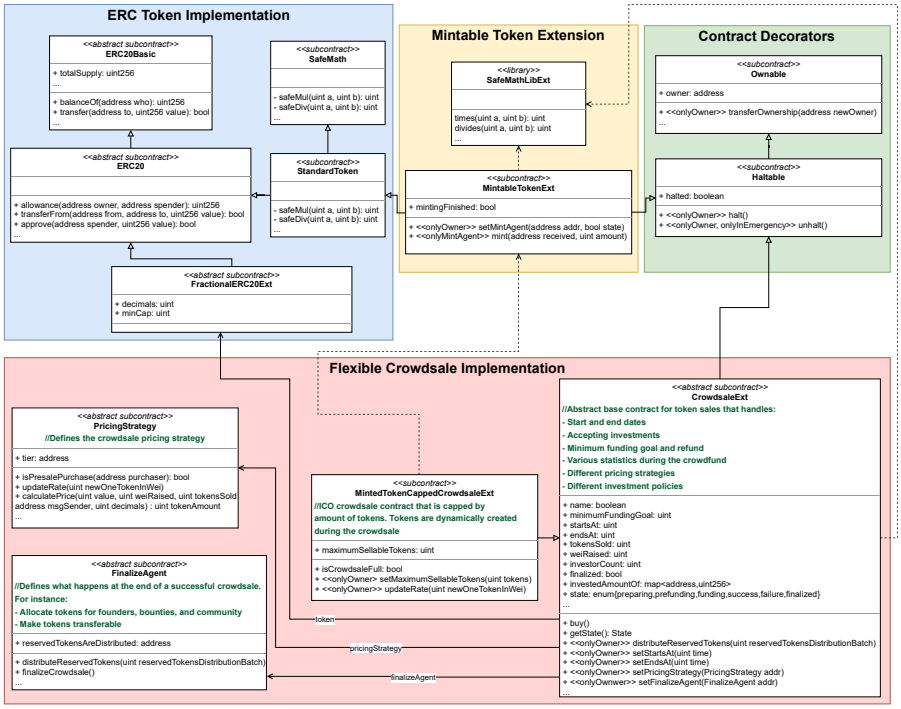

Fig. 12: Class diagram for the code file of the contract deployed at address `0x831979b878dd27f703a19fd1dcaddc2d0b425ac8`. We use colored boxes to group code blocks into cohesive units.

mintable tokens and defines a maximum number of sellable tokens. In particular, the `MintableTokenExt` implements a mintable token by extending the `StandardToken` subcontract. Curiously, the `MintableTokenExt` subcontract uses a different Math library compared to the `StandardToken`.

The two math libraries have overlapping functionalities (e.g., `safeMul(uint a, uint b)` from `SafeMath` and `times(uint a, uint b)`). In the code file, there is a comment that says "*Temporarily have SafeMath here until all contracts have been migrated to SafeMathLib version from OpenZeppelin*", suggesting that `MintableTokenExt` and `StandardToken` were developed in different points in time or by different development teams. The code comment also indicates the preference for an OpenZeppelin version of the library (OpenZeppelin is studied in the RQ3 of this paper).

Finally, the subcontracts in the *Contract Decorators* (green shaded box) define extensions for base contracts. We use the term *decorator* as an allusion to the Decorator object-oriented design pattern (Gamma et al., 1995). The `MintableTokenExt` extends the `Ownable` subcontract, thus becoming decorated with the owning concept. The `CrowdsaleExt`, in turn, extends the `Haltable` subcontract, thus becoming decorated with the `Haltable` concept.

Hence, the `CrowdsaleExt` can be halted and unhalted (by the owner of the crowdsale).

6.1.1.2 On the contract with the highest number of type-2 clone siblings

Similarly to type-1 cloning, type-2 cloning is also transitive. Hence, it suffices to describe any contract in the largest group of type-2 clones. Our analysis revealed that the largest group of type-2 clones contains 1,565 contracts (4.7% of all verified contracts studied in this paper). To serve as a reference, we pick the oldest contract in this group: `0x1a645debd700890f1bc93626078d89e260bd09ce`. Differently from the contract that was discussed in the previous section (Section 6.1.1.1), this contract is very simple: it is a vanilla implementation of a token contract. The UML diagram (drawn by us) shown in Figure 13 illustrates the structure of this token contract. Although declared as a subcontract, the `Token` code block has empty bodied functions, thus fulfilling the role of an interface (as in object-oriented programming). For illustrative purposes, we show the constructor of the `TheMostPrivateCoinEver` contract (Listing 2). From the comments in the code, it is clear that the developer was filling out a template contract.

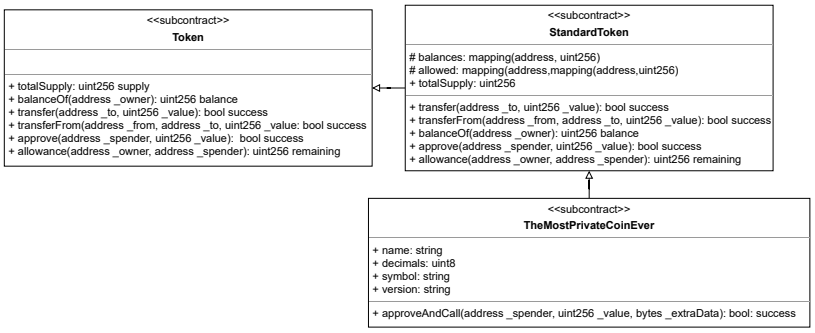

Fig. 13: Class diagram for the code file of the verified contract deployed at address `0x1a645debd700890f1bc93626078d89e260bd09ce`.

```
1 contract TheMostPrivateCoinEver is StandardToken {
2
3   function () {
4     //if ether is sent to this address, send it back.
5     throw;
6   }
7
8   /* Public variables of the token */
9
10  /*
11  NOTE:
12  The following variables are OPTIONAL vanities. One does not have to include them.
13  They allow one to customise the token contract & in no way influences the core functionality.
14  Some wallets/interfaces might not even bother to look at this information.
15  */
16
17  string public name;                 //fancy name: eg Simon Bucks
18  uint8 public decimals;              //How many decimals to show. ie. There could 1000 base units with 3
          decimals. Meaning 0.980 SBX = 980 base units. It's like comparing 1 wei to 1 ether.
19  string public symbol;               //An identifier: eg SBX
20  string public version = 'H1.0';     //human 0.1 standard. Just an arbitrary versioning scheme.
21
22  function TheMostPrivateCoinEver() {
23    balances[msg.sender] = 100000;          // Give the creator all initial tokens
24    totalSupply = 100000;                   // Update total supply
25    name = "The Most Private Coin Ever";    // Set the name for display purposes
26    decimals = 0;                           // Amount of decimals for display purposes
27    symbol = "???";                         // Set the symbol for display purposes
28  }
29
30  ...
31 }
```

Listing 2: `TheMostPrivateCoinEver` subcontract, which implements the `StandardToken` subcontract.

Summary of RQ1

We observed that 79.2% of the studied contracts are clones (i.e., only 20.8% are not a clone of any other contract). In particular:

- There are 10,786 clusters of very similar contracts. However, 20% of the clusters encompass 68% of the verified contracts.
- The percentage of clones among newly created contracts continues to increase over time. In the last quarter of the studied period (2018.Q2), almost 75% of all created contracts were clones.
- 16.7% of the studied contracts are type-1 clones. Type-1 clones have lengthier code files compared to other contracts, indicating that developers fully clone complex contracts. The contract with the highest number of type-1 clone siblings is a complex *run-time configurable contract*.
- 43.3% of the studied contracts are type-2 clones. Type-2 clone pairs commonly have almost identical code files. The few differences tend to be in literals (e.g., coin name, symbol, and supply), contract definitions (e.g., contract name), and function definitions (e.g., function name, function modifiers). The contract with the highest number of type-2 clone siblings is a *development-time configurable contract*.

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

In the previous RQ, we determined the prevalence of cloning in Ethereum. We observed that 79.2% of the studied contracts are clones. Hence, given that developers frequently clone contracts, we now investigate clone clusters more closely. These clusters are the same that we investigated in RQ1, i.e., those produced by Deckard.

We study the following aspects: (i) categories of contracts in large clusters (Section 6.2.1), (ii) activity of contracts within clusters (Section 6.2.2), and (iii) authorship of contracts within clusters (Section 6.2.3). The aforementioned analyses will provide empirical evidence regarding how cloning is performed in Ethereum.

6.2.1 On the categories of contracts in large clone clusters

Motivation. Determining the category of contracts in large clusters should provide insights into the categories of contracts that are more frequently cloned and thus of inherent interest to the smart contract development community.

Approach. We sort clusters according to their size and select the top-10 largest clusters. These top-10 largest clusters account for 19.9% of all the studied contracts. Next, we select a *representative* contract for each of the top-10 clusters, which we define as the contract with the highest number of transactions in the cluster. Based on a manual investigation of the representative contract, we derive a category for the cluster. Given the relevance of token contracts in Ethereum (Section A.4.1), we also determine the percentage of token contracts in each of the top-10 clusters.

Findings. Observation 6) ***9 out of the top-10 largest clusters are token managers.*** As indicated in Table 4, 9 out of the top-10 largest clusters are token managers. A token manager is a token contract that focuses primarily on managing a certain token.

Table 4: A summary of the top-10 largest clusters.

Cluster	Representative Contract		Cluster Size	% of Token Contracts	Cluster Category
	Address	*#Tx.*			
1	0x4672bad527107471cb5067a887f4656d585a8a31	156,956	213	98.6%	Token Manager
2	0x8912358d977e123b51ecad1ffa0cc4a7e32ff774	20,638	217	97.2%	Token Manager
3	0x30392c252da07b69194972e9f770b6dd5deb7af8	1,179	630	99.2%	Token Manager
4	0xa7d7609766d7fcfaf38eda123454bf94b1c1abf0	3	1,097	0.0%	Token Locker
5	0x9064c91e51d7021a85ad96817e1432abf6624470	67,611	486	97.5%	Token Manager
6	0x47a892bf7336a120ee69b2db6acb552acad5f46d	100	320	97.2%	Token Manager
7	0x1d8e5dcf365864fcda8ab39d1f60d66ee2b82770	7,163	610	95.1%	Token Manager
8	0x9a642d6b3368ddc662ca244badf32cda716005bc	308,162	1,933	97.9%	Token Manager
9	0x2eb86e8fc520e0f6bb5d9af08f924fe70558ab89	133,979	378	99.5%	Token Manager
10	0x0a2eaa1101bfec3844d9f79dd4e5b2f2d5b1fd4d	3,264	682	97.1%	Token Manager

Analysis of the structure of the 9 token managers via UML diagrams reveal that they implement very similar functionality in slightly different ways (e.g.,

```
function lock() public onlyOwner returns (bool){
  require(!isLocked);
  require(tokenBalance() > 0);
  start_time = now;
  end_time = start_time.add(fifty_two_weeks);
  isLocked = true;
}

function lockOver() constant public returns (bool){
  uint256 current_time = now;
  return current_time > end_time;
}

function release() onlyOwner public{
  require(isLocked);
  require(!isReleased);
  require(lockOver());
  uint256 token_amount = tokenBalance();
  token_reward.transfer( beneficiary, token_amount);
  emit TokenReleased(beneficiary, token_amount);
  isReleased = true;
}
```

Listing 3: Main functions of the lockEtherPay subcontract. Contract address: `0x0035743c08768ad6558b49d751e0215762057754`.

with different degrees of encapsulation). In the Appendix B.1, we show a UML diagram of the representative contract of each of the top-10 largest clusters.

Observation 7) ***1 of the top-10 largest clusters is a Token Locker.*** The key subcontract from the representative contract of this cluster is called `lockEtherPay`. In Listing 3, we show the key functions of the lockEtherPay subcontract. The `lock()` function checks a few requirement conditions then locks the payment of tokens for 52 weeks. The `lockover()` function checks if the lock period is over. Finally, the `release()` function checks a few requirement conditions, including if the lock period is over, then releases the tokens to the beneficiary. Having a cluster of token lockers suggest that a contract was created to lock the payment of each beneficiary of a certain ICO (or multiple ICOs).

6.2.2 On the activity of contracts within clone clusters

Motivation. The number of received transactions is an indicator of the activity of a contract. We thus investigate this indicator to determine the activeness of the contracts within a given cluster. In particular, we want to determine whether most contracts within a cluster have a similar number of received transactions or whether there is a high degree of variability.

Approach. We use the Gini coefficient to evaluate inequality in the number of transactions per contract in each cluster. The Gini coefficient measures the inequality (i.e., variability) among values of a frequency distribution (Ceriani and Verme, 2012). The coefficient is typically used to evaluate inequality of

income or wealth of a nation's residents. A Gini coefficient of 0% indicates perfect equality, where all values are the same (e.g., a population where everyone has the same income). A Gini coefficient of 100% indicates maximal inequality among values (e.g., a large population in which only one person has all the income and others have none). We compute Gini coefficient of all clusters that contain at least 10 contracts (as it is difficult to judge inequality in clusters that are too small).

Findings. Observation 8) ***Most of the activity of a cluster tends to be concentrated on a few contracts.*** Figure 14 depicts the distribution of the Gini coefficient for the number of received transactions per contract in each cluster. As the figure indicates, the inequality index is generally high for most clusters. In other words, very few contracts within a cluster tend to concentrate most of the activity in that cluster.

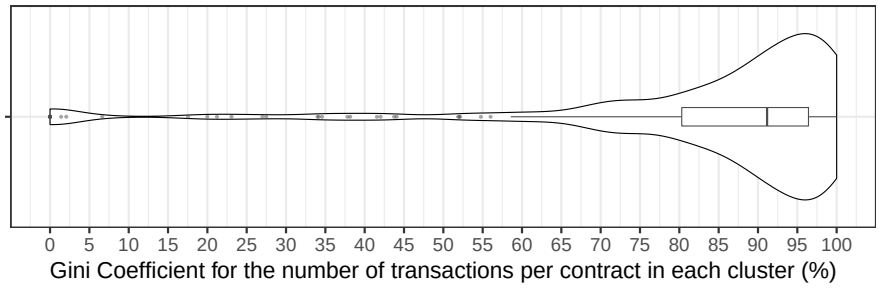

Fig. 14: Distribution of the Gini Coefficient for the number of received transactions per contract in each cluster (Q_1 = 80.3%, median = 91.2%).

Observation 9) ***In 50% of the cases, the top-active contract of a cluster was created before 74.7% of the other contracts in the same cluster.*** We ordered the contracts in each cluster according to their creation date. Next, we recorded the creation date rank of the top-active contract of each cluster. Figure 15 shows the distribution of the creation date rank of the top-active contract of each cluster. Analysis of the figure reveals that, in 50% of the cases, the top-active contracts of a cluster was created before 74.7% of the other contracts in that cluster.

6.2.3 On the authorship of contracts within clone clusters

Motivation. The Ethereum blockchain keeps track of the creator of each smart contract (Section A.3.2). We thus investigate the relationship between authorship and cloning. In particular, we want to discover if there are clusters with a single author (e.g., a developer deploying updated versions of the same contract over time).

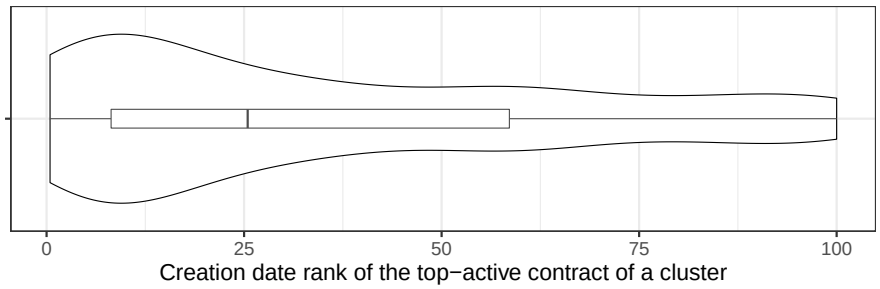

Fig. 15: Distribution of the creation date rank of the top-active contract of each cluster (median = 25.3, Q_3 = 58.3).

Approach. We use the *Normalized Shannon Entropy* (Shannon and Weaver, 1963) to study variability in the authorship of contracts inside a cluster. The Normalized Shannon entropy is formulated as follows:

$$\eta(X) = -\sum_{i=1}^{n} \frac{p(x_i)\log_b(p(x_i))}{\log_b(n)},$$

In our case, X is a cluster, n is the number of contracts in the cluster X, b is the base of the logarithm (we use $b = 2$), x_i is the author of contract i, $p(x_i)$ is the percentage of contracts that were developed by author x_i in the cluster X. $\eta(X)$ is the Normalized Shannon Entropy value for the cluster X, which ranges from zero (one developer) to one (n developers).

As explained by Hassan (2009), for a distribution X where all elements have the same probability of occurrence, i.e., $p(x_i) = \frac{1}{n}$, $\forall i \in 1, ..., n$, we achieve *maximum entropy*. On the other hand, for a distribution X where an element j has a probability of occurrence $p(x_j) = 1$ and $\forall i \neq j : p(x_i) = 0$, we achieve *minimal entropy*.

We compute the Normalized Shannon Entropy in all the clusters that contain at least 10 contracts (as it is difficult to judge variability in clusters that are too small). Afterwards, we use a *quadrant plot* to investigate the relationship between entropy and cluster size. A quadrant plot is simply a scatter plot that divides the cartesian plane in four quadrants by splitting each axis into two parts via some summarizing statistic (typically the median). Figure 16 shows a schematic of a quadrant plot. The rationale is that observations lying on a (sub)set of the quadrants exhibit particular characteristics that are meaningful to the study at hands. In our case, we want to determine if there are large clusters (high cluster size) created by few developers (low authorship entropy).

Findings. Observation 10) ***Contracts in a cluster tend to be created by several authors.*** Figure 17 depicts the distribution of the authorship entropy per cluster. The entropy is generally high, with a median of 81.7% (equivalent to a cluster with 12 contracts, where 4 authors created 2 contracts

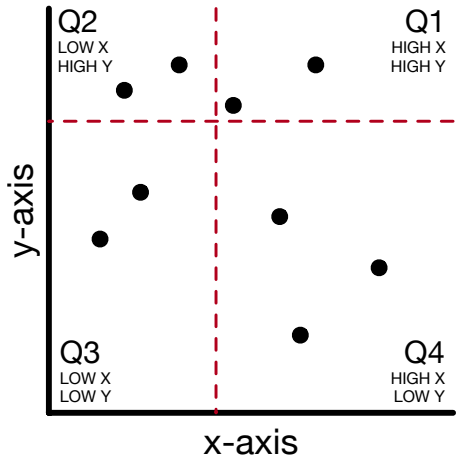

Fig. 16: An example of the quadrant plot. The red dashed lines indicate the median values.

and 4 authors created 1 contract). Therefore, we conclude that contracts in a cluster tend to be created by several authors.

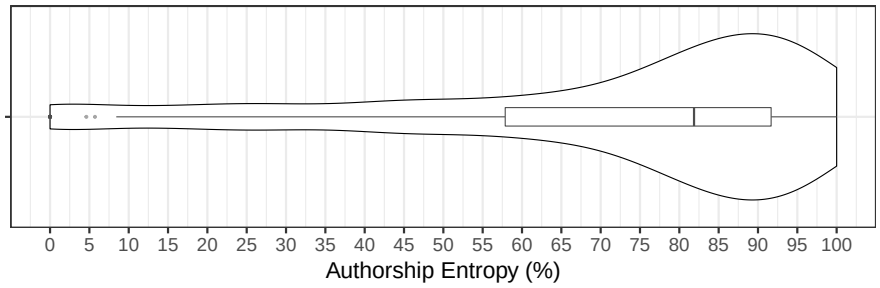

Fig. 17: Distribution of Authorship Entropy (Normalized Shannon Entropy) per clone cluster.

Observation 11) ***There are clusters having many contracts that were created by just a few authors.*** Figure 18 shows the quadrant plot that contrasts cluster size with authorship entropy. As the figure indicates, there are cases in which the entropy is low but the cluster size is high (fourth quadrant). In the following, we discuss two unusual clusters from the fourth quadrant.

At the bottom-right portion of the fourth quadrant (Figure 18), there is an outlier cluster with over 1,000 contracts. Closer inspection reveals that such contracts were created by only five authors. As it turns out, this cluster is the same one that we discussed as part of the top-10 largest clusters analysis: it is

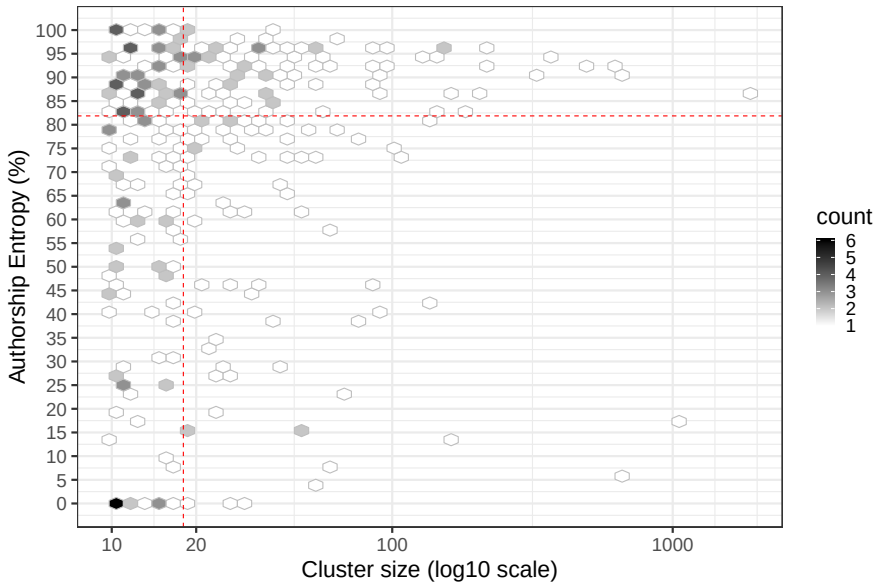

Fig. 18: The quadrant plot of cluster size v.s. authorship entropy. Each hexagon indicates a group of one or more clusters at the same (x,y) position.

the Token Locker cluster (Section 6.2.1). As we discussed before, the authors of these contracts apparently created one contract for each beneficiary of an ICO (or multiple ICOs) and temporarily locked the tokens of each beneficiary. The vast majority of contracts in this cluster (92.9%) are type-2 clones of each other, as the author only changed the beneficiary and the token reward value in each type-2 clone.

We also highlight another cluster in the fourth quadrant, which has zero entropy (i.e., only a single author) and a cluster size of 31. The contracts in this cluster all appear to be scams (Table 5). The author created ICOs that included tokens with the same name as hot tokens being sold at that time. In addition, as shown in Table 5, all contracts in this cluster were created on Christmas (or very close to it, depending on the timezone). The intent was probably to quickly steal cryptocurrency from inexperienced users by pointing them to these *probable-fake* ICOs.

For instance, the real ICO of the EthLend Token was a success and sold all tokens, accumulating 17.9M USD worth of cryptocurrency (its probable-fake version is listed with ID 30 in Table 5). Figure 19 shows the top portion of the code file of the probable-fake EthLend ICO contract. The header of the code file contains links to the real EthLend website and white paper.

[9] `https://etherscan.io/address/0x6b31a898f7e711b323a6212eac4ae250e0d6624f#code`

Table 5: Cluster of clone contracts created by the single author `0x00a7ca00471d62dece1b52ab409b8307836072c3`. The listed contracts all appear to be scam ICOs, since they reuse the name of famous tokens that were being sold at the time.

ID	Contract Address	#Tx.	Token	Creation Date (yyyy-mm-dd)	Info about Real Token
1	0x2066a2e0cd7f19589f582e2fc9af9669cc9e02f1	6	Sether (SETH)	2017-12-25 01:16:50	https://icobench.com/ico/sether
2	0xefe14cde6752b34693e368dba320f4f1897d0556	7	GIFTO (GIFTO)	2017-12-25 01:38:02	https://icodrops.com/gifto
3	0x353230862ae8ab6ca5dde592654acd6cdec1c1b2	6	Beetoken (BEE)	2017-12-25 01:43:56	https://icodrops.com/the-bee-token
4	0xe50b0cefeb80bdd9e3d035179769097 65c96e962	6	Coinvest (COIN)	2017-12-25 01:43:56	https://icodrops.com/coinvest
5	0x828be8ea7bbbe246216e7017b35057f64cf6449a	5	ModulTrade (MTRc)	2017-12-25 01:50:43	https://icobench.com/ico/modultrade
6	0x0f771aa18c5003aba1b0feee082a0dd6acd29956	7	JibrelNetworkToken (JNT)	2017-12-25 01:52:46	https://icodrops.com/jibrel-network
7	0x301a0501c08e347b062b3c928fd4d0155f367855	7	WePower (WPR)	2017-12-25 01:52:46	https://icodrops.com/wepower
8	0x9432f91bfc2335b0485aa4967d0acff3e34ae134	10	MedToken (MED)	2017-12-25 01:54:27	https://icodrops.com/medicalchain
9	0x42b03689fbaa9635edbc10ad1fdc7ea5feefbb70	8	BitDegree (BDG)	2017-12-25 01:55:36	https://icodrops.com/bitdegree
10	0x82bb5dcd77686587633ba6744bd706ea09833fbf	9	DMarket Token (DMT)	2017-12-25 01:57:28	https://icodrops.com/dmarket
11	0xd4fac84b747aba53fd86a9951d20e7330daa3414	13	Powerledger Token (Powr)	2017-12-25 02:02:53	https://icodrops.com/powerledger
12	0x1a51f16b4d385a06c542d3686412ab6517b5515d	7	DomRaider (DRT)	2017-12-25 02:15:30	https://icodrops.com/domraider
13	0x51489a328c9bdfcb67cce77c0d87be54b5f8d172	7	UTRUST Token (UTK)	2017-12-25 02:45:07	https://icodrops.com/utrust
14	0x69c9e34a64b6ba1e39fb15ffaefdc7e75524f600	7	Gimli Token (GIM)	2017-12-25 02:49:52	https://icodrops.com/gimli
15	0x1cf6949f4b661018b3195f3ff98effe12f587263	8	Atlant Token (ATL)	2017-12-25 02:52:00	https://icodrops.com/atlant
16	0xb09bd0cda4c4254acebc130ca17915b93519079c	12	SIRIN (SRN)	2017-12-25 02:52:11	https://icodrops.com/sirin-labs
17	0xabcaea6f4099b1ed62169d39e26f808f948c86d6	5	Decentraland Mana (MANA)	2017-12-25 02:53:02	https://icodrops.com/decentraland
18	0x9d0a108fbb9b311e9976bc071d446912365956 2f	10	Lendoit (LDI)	2017-12-25 02:53:06	https://icodrops.com/lendoit
19	0x38e308c4ac50fd6c31c5f53d7b25f25116d80a3a	6	Scrinium (SCR)	2017-12-25 02:55:26	https://icobench.com/ico/scrinium
20	0x818ebdad12d3decefabe6ccb16f33c9bbfe1b30a	5	NAGA Coin (NGC)	2017-12-25 02:56:21	https://icodrops.com/naga
21	0x2bf36e414c52a9fd81e21a0ab5742dce750389d3	8	BankEx Token (BKX)	2017-12-25 03:04:06	https://icodrops.com/bankex
22	0x8446563e75cd7821b5bf2199c26976efc5cac803	14	COVESTING (COV)	2017-12-25 03:36:25	https://icodrops.com/covesting
23	0x889a32263d237872b4f7985ead9c29ebfe8b464d	13	Caviar Token (CAV)	2017-12-25 03:36:38	https://icobench.com/ico/caviar
24	0x50377c3314581a639f00bd5ba42367a6b65bced7	14	DreamTeam Token (DTT)	2017-12-25 03:39:23	https://icodrops.com/dream-team
25	0x11838ece89439c4f0224bc7b9f76e07721d99f9e	13	MiCars Token (MCR)	2017-12-25 03:42:50	https://icobench.com/ico/micars
26	0x928989d4d0d71daaf86819ca69df620ebaaad407	14	Majato Token (MJT)	2017-12-25 03:48:00	https://icobench.com/ico/majato-project
27	0x6fdfce60998725f4cfb4cc57505a3abdbb6d0646	17	SelfKey Token (KEY)	2017-12-25 04:01:33	https://icodrops.com/selfkey
28	0x155e7e9ab193aa37c87d22f66002e3db64e9e0ce	15	Props Token (PROPS)	2017-12-25 04:13:15	https://icodrops.com/props
29	0xfda2e0437f974c9312ca6740c07181178f645fa9	14	QLink Token (QLC)	2017-12-25 04:13:27	https://icodrops.com/qlink
30	0x6b31a898f7e711b323a6212eac4ae250e0d6624f	13	EthLend Token (LEND)	2017-12-25 04:27:49	https://icodrops.com/ethlend
31	0x2920e0b7d86ab76aa09bcddffedef081faaf6e2c	9	Bloom Token (BLT)	2017-12-25 04:33:21	https://icodrops.com/bloom

Contract Source Code </>

```
pragma solidity ^0.4.19;


/// @title  EthLend Token internal presale - https://ethlend.io (LEND) - crowdfunding code
/// Whitepaper:
///  https://github.com/ETHLend/Documentation/blob/master/ETHLendWhitePaper.md

contract EthLendToken {
    string public name = "EthLend Token";
    string public symbol = "LEND";
    uint8 public constant decimals = 18;
    address public owner;

    uint256 public constant tokensPerEth = 1;
    uint256 public constant howManyEtherInWeiToBecomeOwner = 1000 ether;
    uint256 public constant howManyEtherInWeiToKillContract = 500 ether;
    uint256 public constant howManyEtherInWeiToChangeSymbolName = 400 ether;

    bool public funding = true;

    // The current total token supply.
    uint256 totalTokens = 1000;

    mapping (address => uint256) balances;
    mapping (address => mapping (address => uint256)) allowed;
```

Fig. 19: The header of the code file of the fake contract contains links to the real EthLend website and white paper. Image extracted from Etherscan[9].

We also observe that the author sent transactions himself to the contracts, possibly to give more credibility to the probable-fake ICOs and entice others to use them. An example is depicted in Figure 20.

[10] `https://etherscan.io/address/0x6b31a898f7e711b323a6212eac4ae250e0d6624f`

TxHash	Block	Age	From		To	Value	[TxFee]
0xd3da2f1db9a3416...	4793059	444 days 9 hrs ago	0x00a7ca00471d62...	IN	0x2066a2e0cd7f195...	0 Ether	0
0xc140961d7aa26fc...	4793056	444 days 9 hrs ago	0x00a7ca00471d62...	IN	0x2066a2e0cd7f195...	0 Ether	0
0x94aa52bc4e4ccb...	4793056	444 days 9 hrs ago	0x00a7ca00471d62...	IN	0x2066a2e0cd7f195...	0 Ether	0
0x318559fe20049f4...	4793056	444 days 9 hrs ago	0x00a7ca00471d62...	IN	0x2066a2e0cd7f195...	0 Ether	0
0x6f68f0e49694d22...	4793056	444 days 9 hrs ago	0x00a7ca00471d62...	IN	0x2066a2e0cd7f195...	0 Ether	0
0x1bfa5dcddabac6c...	4792647	444 days 11 hrs ago	0x00a7ca00471d62...	IN	0x2066a2e0cd7f195...	0 Ether	0
0x03308329334be1...	4791691	444 days 14 hrs ago	0x00a7ca00471d62...	IN	Contract Creation	0 Ether	0.000279430625

Fig. 20: Author of the probable-fake Sether contract sent 6 transactions himself to the contract (check "From" field). Image extracted from Etherscan[10].

Summary of RQ2

In this RQ, we investigated three key characteristics of clone clusters, namely: category (Section 6.2.1), activity concentration (Section 6.2.2), and authorship (Section 6.2.3). We observed that:

- 9 out of the top-10 largest clusters are token managers. The other cluster contains Token Locker contracts, which lock the tokens of a given beneficiary for a predefined amount of time.
- Most of the activity of a cluster tends to be concentrated on a few contracts (Q1 and median of the Gini coefficients for activity within a cluster are 78.34% and 90.72% respectively.)
- In 50% of the cases, the top-active contract of a cluster was created early on (before 74.7% of the other contracts in the same cluster)
- Contracts in a cluster tend to be created by several authors (median entropy of 81.7%). Yet, there are exceptional cases of clusters having many contracts that were created by just a few authors. In particular, there is a cluster with 31 probable-scam contracts that were created by a single author.

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

Motivation. OpenZeppelin is one of the most popular packages for developing 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 February 25th 2019, the project has 1,518 commits, 163 contributors, and 6,791 stars in its GitHub repository. The development team behind OpenZeppelin aims to produce high-quality code to be reused by smart contract developers. The team claims to adhere 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. As such, these code blocks are meant to be reused without modification. In this vein, we study the code blocks of verified contracts that are identical to those from OpenZeppelin.

Approach. Code files of verified contracts typically contain several code blocks. In turn, each OpenZeppelin code file contains a single code block. Therefore, we first extract the code blocks from the code file of each verified contract. Next, we determine the identical blocks to those from OpenZeppelin. In the following we describe the approach in more details:

1) Extracting code blocks from the code file of verified contracts. We used the Solidity parser created by Federico Bond (Section 3) to extract the code blocks from the code files of verified contracts. Given a code file, the parser outputs the beginning and ending position of each code block in the file. The parser successfully extracted the code blocks from 32,465 (98.16%) of the code files. In the remaining cases, an error was thrown by the parser. A total of 165,005 code blocks were found.

2) Extracting code blocks from OpenZeppelin. As shown in Table 1, we downloaded 13 OpenZeppelin releases (since new releases include new code blocks). Each OpenZeppelin release contains a set of code files and each code file contains exactly one code block. Therefore, the extraction of code blocks from OpenZeppelin is trivial.

3) Determining the identical code blocks between the code blocks from the code file of verified contracts and the code blocks from OpenZeppelin. For each code block from the code file of a verified contract, we determine the code block from OpenZeppelin that it is identical.

4) Determining the category of an OpenZeppelin code block. As part of this RQ, we want to determine which categories of OpenZeppelin code blocks have the higher number of identical code blocks. We determine the category of an OpenZeppelin code block according to its path in the repository. For instance, we say that the `math/SafeMath.sol` code file belongs to the *math* category. If a code file lies in the root directory, its category is unknown (UNK).

Newer versions of OpenZeppelin have more descriptive file paths compared to older versions. For example, the path of the `FinalizableCrowdsale.sol` code block in version 1.1.0 is `crowdsale/FinalizableCrowdsale.sol`. In turn, the path of this same code block in version 1.12.0 is `crowdsale/distribution/FinalizableCrowdsale.sol` (i.e., more descriptive).

```
1  n <- number of releases from OpenZeppelin
2  for i from 1 until (n-1) do {
3    //releases are sorted in chronological order
4    old_release <- get release i;
5    old_files <- extractFiles(old_release);
6    for each old_file in old_files do {
7      for j from n until (i+1) do {
8        new_release <- get release j;
9        new_files <- extractFiles(new_release);
10       if the basename of old_file matches the basename of a new_file in
             new_files do {
11         override(old_file.category, new_file.category)
12         break
13       }
14     }
15   }
16 }
```

Listing 4: Algorithm for overriding categories of code files from older releases of OpenZeppelin with categories of code files from newer releases of OpenZeppelin.

Hence, to obtain more descriptive categories for code files, we override the category of code files from older OpenZeppelin releases with the category from newer OpenZeppelin releases. The algorithm that we conceived is described in Listing 4. For instance, assume that a certain code block c from a verified contract is identical to the `crowdsale/FinalizableCrowdsale.sol` code block that was released in version 1.1.0. After running our algorithm, we can then say that c is actually identical to a *crowdsale/distribution* code block (instead of a *crowdsale* code block).

Finally, we highlight that our approach does not aim to establish clone genealogies (e.g., determine whether the developer of a certain verified contract wrote a code block by copying it from OpenZeppelin). For the goal of this study, it suffices to simply detect all pairs of the form $\langle C_e, C_o \rangle$, where C_e is a code block from a verified contract on Etherscan, C_o is a code block from OpenZeppelin, and C_e and C_o are identical.

Findings. Observation 12) ***36.3% of the verified contracts have at least one code block in their code file that is identical to an OpenZeppelin code block.*** In Figure 21, we show the percentage of verified contracts that have a certain percentage of code blocks that are identical to OpenZeppelin code blocks. As indicated by the green bar from the (10,0) percentage range, 36.3% of the verified contracts have at least one code block that is identical to an OpenZeppelin code block. In addition, the green bar from the (60,50] percentage range indicates that 50% of the code blocks from 18.0% of the verified contracts are identical to OpenZeppelin code blocks.

Observation 13) ***26.3% of all 165,005 code blocks extracted from verified contracts are identical to OpenZeppelin code blocks. Most part of these code blocks belong to the ERC20 OpenZeppelin category.*** Figure 22 shows the percentage of identical code blocks per OpenZeppelin cat-

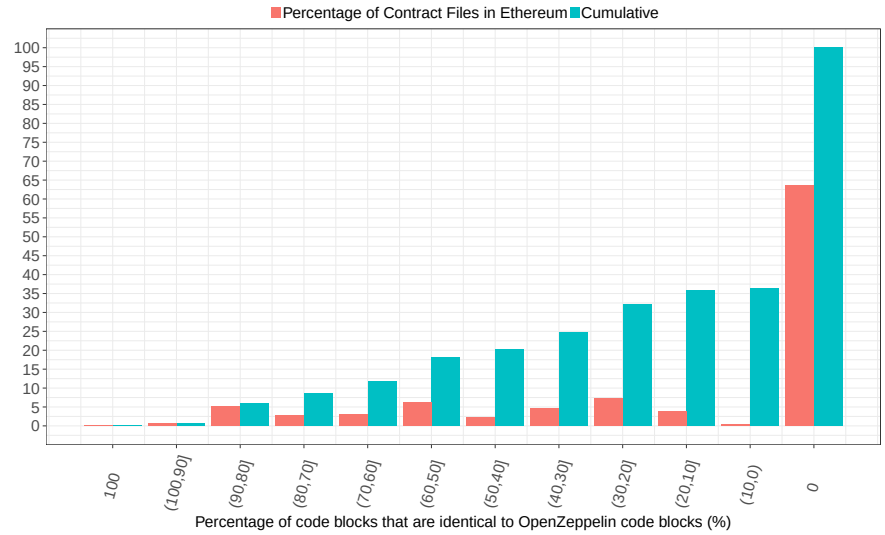

Fig. 21: Bar chart showing the percentage of verified contracts that have a certain percentage of code blocks that are identical to OpenZeppelin code blocks.

egory. Most code blocks belong to the *ERC20* category, which comprises code blocks that relate to the ERC20 standard (Section A.4.1). More specifically, 15.4% of all 165,005 code blocks extracted from verified contracts are identical to ERC20 code blocks. The second most identical category is *math*, which includes utility code blocks that support mathematical operations. The third most identical category is *ownership*, which contains code blocks that work as decorators for existing contracts, given them the ability to be owned by a certain Ethereum account. The forth most identical category is *lifecycle*, which includes code blocks that enable the pausing and unpausing contract. The *others* category serves as a bundle that groups all other categories, namely: *validation*, *crowdsale*, *distribution*, *ERC721*, *UNK*, *ERC827*, *utils*, *emission*, *rbac*, *payment*, *token*, *access*, *price*, and *mocks*. As the green bars indicate, 26.3% of all 165,005 code blocks extracted from verified contracts are identical to OpenZeppelin code blocks.

Observation 14) ***SafeMath.sol, ERC20.sol and ERC20Basic.sol are the most frequently reused code blocks from OpenZeppelin.*** Figure 23 is analogous to Figure 22, but shows code blocks instead of categories. The first most frequently reused code block is `SafeMath.sol`, which belongs to the math category. This code block contains functions that perform mathematical operations efficiently and without incurring into under/overflow. The second and third most frequently reused code blocks are `ERC20.sol` and `ERC20Basic.sol`, which belongs to the ERC20 category. These code blocks support the implementation of ERC20 token contracts. The fourth most fre-

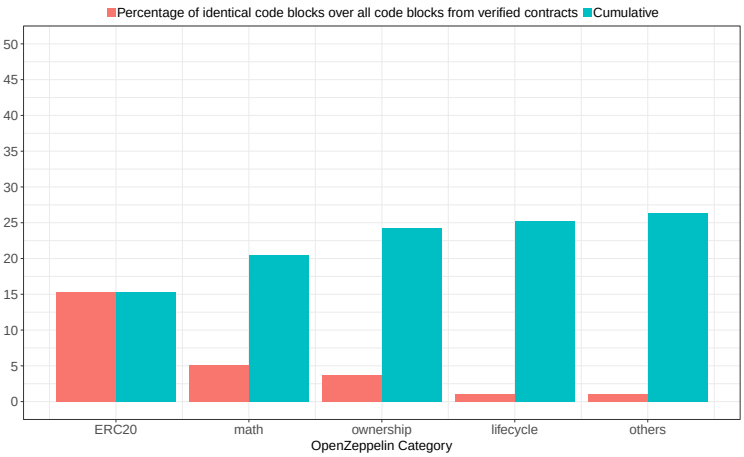

Fig. 22: Percentage of identical code blocks per OpenZeppelin category.

quently reused code block is `Ownable.sol`, which adds an owner to a contract and provide basic authorization control functions. The fifth and sixth most frequently reused code blocks are earlier code blocks that also supported the implementation of ERC token contracts. The seventh most frequently reused code block is `Pausable.sol`, which enables contracts to implement an emergency stop mechanism via a `pause()` function. Finally, the eighth most frequently reused code block is `MintableToken.sol`, which offers an implementation of a mintable token (Section A.4.2).

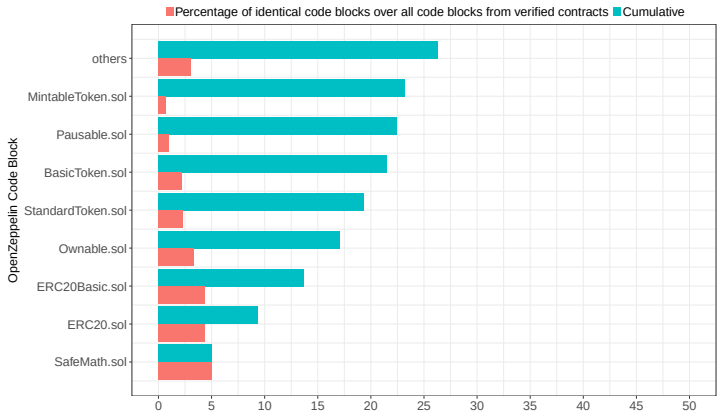

Fig. 23: Percentage of identical code blocks per OpenZeppelin code block.

Summary of RQ3

About one third of all 165,005 code blocks extracted from verified contracts are identical to OpenZeppelin code blocks. In particular:

- 36.3% of the verified contracts have at least one code block in their code file that is identical to an OpenZeppelin code block.
- 50% of the code blocks from 26.3% of the verified contracts are identical to OpenZeppelin code blocks.
- The ERC20 OpenZeppelin category is the most frequently reused category, which contains code blocks to support the implementation of token contracts that comply with the ERC20 standard.
- `SafeMath.sol` is the most frequently reused OpenZeppelin code file, which contains functions that perform mathematical operations efficiently and without incurring into under/overflow.

7 Discussion

7.1 Implications

7.1.1 Implications to the Security of Smart Contracts

Implication 1) ***The impact of exploiting a smart contract vulnerability in Ethereum is large.*** Observation 2 indicates that there are large clone clusters in Ethereum (20% of the contracts encompass approximately 68% of the contracts.). In particular, Observation 6 indicates that 9 out of the top-10 largest clusters are token contracts, meaning that these contracts hold tokens that have an associated market capitalization (in addition to any Ether balance that the contracts might hold). Therefore, if a hacker manages to discover and exploit a vulnerability in one of the contracts belonging to these large clusters, it is possible that the hacker would be able to exploit many of the cloned versions of the vulnerable contract. Such a threat is aggravated by three reasons:

– 16.7% of the studied contracts are type-1 contracts (Observation 4). In other words, an exploit that works for any of these contracts will also likely work for at least one more contract. In addition, type-1 contracts have lengthier code files, thus are more susceptible to vulnerabilities. The contract with the highest number of type-1 siblings is particularly complex (Section 6.1.1.1).

– 43.3% of the studied contracts are type-2 contracts (Observation 5). We also note that type-2 clone pairs are remarkably similar. Hence, similarly to the type-1 case, we conjecture that the chances of being able to exploit the same vulnerability in multiple contracts is high.

– Observation 3 indicates there is an upward trend in the amount of clones being created every quarter.

We reiterate that the code of smart contracts cannot be changed once they are deployed in Ethereum, so the vulnerability cannot be fixed by a code change. Moreover, due to the immutability of a blockchain, previous blocks cannot be altered after they are deemed final (which takes approximately 3 minutes after the block is added to the blockchain). Hence, as opposed to a traditional database system in which transactions can be somewhat easily reverted, a transaction inside a final block is also final. In other words, a hacker's actions can only be reverted by means of a non-planned *hard fork*, which is a radical procedure. In simple terms, a hard fork means creating another version of the blockchain, where all nodes in the network are required to upgrade to the latest version of the protocol software in order to make previously valid blocks invalid (or vice-versa). For instance, the famous "DAO" incident described in the Introduction (Section 1) resulted in a hard fork and raised concerns in the community regarding the credibility of the whole Ethereum platform.

Implication 2) ***The absence of OpenZeppelin code blocks in 64% of the studied contracts poses questions about the vulnerability of such contracts.*** As we discussed above, due to the amount of clones, the impact of exploiting a smart contract vulnerability in Ethereum is large. A natural question that derives from such a conclusion is: how vulnerable are smart contracts? As we describe in the motivation of RQ3, OpenZeppelin is a project devoted to the creation of secure and reusable code blocks for smart contracts. Hence, we assume that smart contracts that reuse code blocks from OpenZeppelin are less prone to having vulnerabilities (or at least well-known, high-risk vulnerabilities).

In RQ3, Observation 12 indicates that approximately 36% of the studied contracts have at least one code block in their code that is identical to an OpenZeppelin code block. In particular, for 18% of the studied contracts, at least 50% of their code blocks are identical to those provided by OpenZeppelin. Notably, Observations 13 and 14 indicate that these code blocks frequently relate to the implementation of the ERC20 standard (`ERC20.sol` and `ERC20Basic.sol`) and mathematical operations (`SafeMath.sol`). Therefore, developers seem to prioritize the use of secure code for basic token management operations, as well as mathematical operations.

An orthogonal analysis perspective indicates, however, that approximately 64% of the studied contracts do not have any code block in common with those from OpenZeppelin. Further research is required to determine whether the code blocks of these contracts are actually safe (e.g., through a combination of code analyzers ((di Angelo and Salzer, 2019)) and auditing) and/or reuse code blocks from other reputable sources (e.g., because they tackle an application domain that is not covered by OpenZeppelin).

7.1.2 Implications to the Development of Smart Contracts

Implication 3) ***Cloning is a common practice in Ethereum. Yet, cloning a contract does not imply it will achieve the same popularity.*** In Observation 1, we note that only 20.8% of the verified contracts are not a clone of any verified contract. Hence, cloning is a common practice. Given the prevalence of cloning, in RQ2 we studied the characteristics of clone clusters. Observation 8 indicates that most of the activity of a cluster tends to be concentrated on very few contracts (Figure 14). In other words, each cluster seems to have only a small group of contracts that attract attention. In addition, Observation 9 indicates that in 50% of the cases, the top-active contract of a cluster was created early on (before 3/4 of the other contracts in the same cluster). Finally, Observation 10 notes that contracts in a cluster tend to be created by several authors. We thus conclude that developers clone highly-active contracts, yet these clones rarely achieve the same activity level. This result is relevant from a practical perspective, since contracts with high-activity have historically enticed other developers to clone them. For instance, the booming success of the Ethereum-powered game CryptoKitties[11] (a game in which players collect and breed digital cats) in the late 2017 led to the development of a plethora of Chinese clone versions, such as CryptoDogs(Horwitz and Huang, 2018) and CryptoAlpaca(Skvorc, 2018). However, neither of these clones ever achieved the same popularity of CryptoKitties.

Implication 4) ***Development of a significant portion of the studied contracts is template-driven. Developers should instead rely on configurable contracts.*** We note from Observation 5 that 43.3% of the studied contracts are type-2 clones. In other words, many contracts only differ in terms of identifiers and literals. Therefore, the development of a significant portion of verified contracts in Ethereum is template-driven. Indeed, projects such *ConsenSys Tokens*[12] provide minimalist token implementation templates to be reused by smart contract developers. Upon manual inspection, we noticed that the largest type-2 cluster that we identified (Section 6.1.1.2) actually builds on a template[13] provided by the ConsenSys Tokens project.

Code cloning via templating is a questionable practice, since developers need to manually change the source code (e.g., variable values regarding token name and supply) every time a new concrete implementation needs to be deployed. Instead, developers should opt for *configurable contracts*. Instead of hardcoding values in the code, configurable contracts allow variables be set up at contract deployment time (e.g., via the contract constructor) or at runtime (e.g., through set functions). Such an approach allows values to be type-checked and requires zero code changes for new variations of the origi-

[11] `https://www.cryptokitties.co`

[12] `https://github.com/ConsenSys/Tokens`

[13] `https://github.com/ConsenSys/Tokens/blob/master/contracts/eip20/EIP20.sol`

nal contract. All code blocks from the OpenZeppelin are designed with this philosophy (configuration over manual change).

Finally, given the prevalence of type-2 cloning, the development team behind Solidity could consider adding template functionalities to the language (similarly to C++ templates).

Implication 5) ***Providers of reusable code blocks should explicitly indicate provenance in the source code.*** It is difficult for end-users to determine whether the code file of a verified contract is of good quality just by glancing through it. In particular, as our results indicate, the code files of verified contracts have varying ratios of OpenZeppelin code blocks (Figure 21). Hence, projects such as OpenZeppelin should consider adding provenance information directly in the source code. For instance, a code comment header could be added to their code blocks, which would include an explicit mention to the OpenZeppelin project and the corresponding version of the code block. A suggestion is shown below:

```
/**
* @origin This subcontract was developed by the OpenZeppelin team.
* Location: contracts/token/ERC20/ERC20.sol. Version: v1-12-0
*/
```

Since the code file is a flattened version of the original source code (Section A.3.3), we would also encourage OpenZeppelin to include delimiters at the beginning and end of their code blocks. With these two measures in place (header and delimiters), end-users would know exactly which code blocks of a verified contract were reused from OpenZeppelin and their corresponding versions.

We have suggested this change to the OpenZeppelin team by opening an issue in their GitHub repository. Currently, the opened issue is under discussion[14,15].

Implication 6) ***Despite the prevalence of clones, Ethereum does not offer any mechanisms for developers to easily track contract upgrades.*** Assume a developer D_1 deployed a contract C. Assume as well that a developer D_2 cloned C and deployed it. Now, consider that a severe bug is found on C and a new version with the fix is made available by D_1. Let us call the new contract version C_{fix}. As of today, there is no mechanism to alert D_2 (and the user base of C) that (i) a severe bug has been found on C and that (ii) a new version with a fix exists (i.e., C_{fix}). At the simplest level, solving the problem would involve tracking contract creation dates, cloning, and new releases of clones. Currently, popular Ethereum dashboards such as Etherscan are only able to inform whether a given contract has clones. A solution at a more sophisticated and general level would likely need to tackle the following requirements: (i) contracts should be versioned (e.g., OpenZeppelin has

[14] https://github.com/OpenZeppelin/openzeppelin-contracts/issues/1716
[15] https://github.com/OpenZeppelin/openzeppelin-contracts/issues/2006

recently decided to adopt the *SemVer* semantic versioning scheme[16,17]), (ii) such versioning needs to be tracked by a central package manager (e.g., similarly to Maven[18] or npm[19]), and (iii) the package manager is able to inform whether a new version of a contract is available and its changelog. Non-official package managers for Ethereum are currently under development[20].

7.2 Avenues for Future Research

– In this study, we only detected type-1 and type-2 clones. Despite the inherent challenge, detecting type-3 clones would provider a richer view of cloning and code reuse practices for Ethereum.

– In this study, we used Deckard to detect code clones, which operates at the source code level. The code cloning research community has developed several other detectors that operate at the source code level, such as iClones (Göde and Koschke, 2009), NiCad (Cordy and Roy, 2011), CCFinderX (Kamiya et al., 2002), and SourcererCC (Sajnani et al., 2016). Recently, clone detectors that operate at the bytecode level have also been proposed (Liu et al., 2019). Comparing the results yielded by all these tools should provide insights into the advantages and drawbacks of each one (e.g., in terms of accuracy and processing time).

– Observation 4 notes that 16.7% of the contracts that we studied are type-1 clones (i.e., they are identical to some other contract) and they have lengthier code files compared to other contracts. We find it rather surprising that big pieces of code are being reused as is, without the need for modifications. We conjecture that such contracts tend to be configurable contracts (Implication 4). Indeed, in Section 6.1.1.1, we noted that the type-1 contract with the highest number of siblings is a runtime configurable contract. Such a conjecture requires further and deeper investigation.

– Code cloning literature indicates that cloning can be either beneficial or harmful. For example, cloning a complex contract from a reputable source (e.g., OpenZeppelin) is likely beneficial compared to writing it from scratch. Future work is required in order to categorize smart contract cloning practices and uncover their impact.

– We observed that cloning is extremely prevalent in Ethereum. A qualitative study involving smart contract developers (e.g., a survey, an observational study, or a sequence of interviews) would provide additional insights into whether cloning in Ethereum is incidental or intentional.

[16] https://semver.org
[17] https://docs.openzeppelin.com/contracts/2.x/api-stability
[18] https://maven.apache.org
[19] https://www.npmjs.com
[20] http://ethpm.com

– Other programmable blockchains such as EOS[21] and POA[22] are starting to become more popular. Cloning practices in these other blockchain platforms remain unknown and should be examined in follow up studies.

8 Related Work

Code Cloning. There is a vast literature in Software Engineering around the code cloning theme (Roy and Cordy, 2007; Koschke, 2008; Sheneamer and Kalita, 2016). The conflicting views on the harmfulness of code clones (e.g., compare (Kim et al., 2005; Kapser and Godfrey, 2008; Bettenburg et al., 2012) with (Kamiya et al., 2002; Bellon et al., 2007; Juergens et al., 2009)) indicate that conclusions depend on the context in which the software system is developed and the perspective taken in the research study (e.g., developer vs end-user).

There are only a few studies regarding code clones in Ethereum. The most similar study to ours is the work of Gao et al. (2019), who also identify code clones from the source code of smart contracts. The authors developed their own detection tool called SmartEmbed, which relies on code embeddings (Bojanowski et al., 2017) and similarity checking techniques. The goal of their tool is to quantify clones in Ethereum and identify clone-related bugs. They investigated 22k contracts and determined that the clone ratio is close to 90% (measured at the line level). In our study, we found a clone ratio close to 80%. Moreover, we analyzed clone prevalence in more detail by observing clone types, trends, and contextual information about clones (e.g., what do they do, who creates them, how active they are). The scale of our study is 48% larger: we investigated a total of 33,073 smart contracts. Gao et al. (2019) state that their tool can detected 194 clone-related bugs with a precision of 96%.

Still in the domain of Ethereum, Liu et al. (2018, 2019) proposed a clone detector that operates at the bytecode level. Given that Ethereum stores the bytecode of every deployed smart contract, such an approach has the potential to detect clones among any set of currently deployed smart contracts on Ethereum. However, detecting clones at the bytecode level is inherently more challenging compared to detecting them at the source code level. For instance, a different bytecode can be produced for the same source code depending on the version of the used compiler and the selected optimization parameters of that compiler. To circumvent these problems, the authors introduce the notion of *smart contract birthmarks*. According to the authors, a birthmark is a semantic-preserving and computable representation for smart contract bytecode. This birthmark relies primarily on symbolic execution traces and maintains syntactic properties of the code (e.g., number of instructions). Clone detection is then performed by comparing the birthmarks of two contracts. The authors report precision scores in the range of 54% to 93%. As opposed to Liu et al. (2018, 2019), our goal is *not* to develop a new clone detector for

[21] `https://eos.io`
[22] `https://poa.network/`

smart contracts. Instead, we employ a mature clone detector that operates at the source code level (Deckard) and focus on a deeper understanding of the detected clones.

Ethereum can be seen as a repository of software applications (smart contracts). Therefore, it makes sense to compare the cloning ratios that we obtained with those found in cross-project cloning studies. In the context of GitHub, Lopes et al. (2017) discovered that 70% of the code on GitHub consists of clones of previously created files. This ratio is in a similar range as the one obtained by us (79.2%), though much lower than the reported ratio by Gao et al. (2019) of 90%. Lopes et al. (2017) highlights that researchers should be aware of the actual lack of project diversity in GitHub (especially for the JavaScript language). We issue a similar warning. Despite the hype around blockchain technology and its supposedly wide range of use cases, what we actually see is a similar lack of diversity.

Lastly, we note that very high cloning rates have also been found in other studies. For instance, Mockus (2007) analyzed a vastly heterogenous sample of open-source projects and observed that approximately 50% of the files were used in more than one project (type-1 clones). Comparatively, of all contracts that we studied, 16.7% were classified as type-1 clones. The key difference compared to our context, however, is that smart contracts cannot be modified once deployed. Hence, while bugs can be fixed in traditional software development through patches, such bugs cannot be fixed via code changes in Ethereum.

Immutability of Smart Contracts. Code cloning in Ethereum is intrinsically related to the immutability property of smart contracts. Fröwis and Böhme (2017) investigated the immutability of the control flow of smart contracts by means of static analysis. According to the authors, not only code immutability is necessary for trustlessness, but also control flow immutability (i.e., call relationships between contracts should not change on runtime). To find immutability violations, the authors extracted call relationships between smart contracts and searched for contract addresses that are provided as input parameters or that are read from state variables. The authors concluded that two out of five smart contracts require trust in at least one third party.

Security of Smart Contracts. Since blockchains applications typically operate on cryptocurrencies, there is a large concern in the communities of both researchers and practitioners around the security aspect of smart contracts. Such a concern became even more relevant after the incident with "The DAO" and the consequent Ethereum hard-fork. Luu et al. (2016) wrote a symbolic execution tool called OYENTE[23] to find potential security bugs. According to the authors, the tool flagged 45.6% of the contracts in Ethereum as potentially vulnerable. Kalra et al. (2018) also leverage symbolic execution to verify the correctness and fairness of smart contracts. Correctness is defined as adherence to safe programming practices, while fairness is adherence to agreed upon higher-level business logical (i.e., does the contract do what the author

[23] `https://github.com/melonproject/oyente`

says it does?) The fairness evaluation is the main novelty compared to prior work. According to the authors, 94.6% of the smart contracts are vulnerable to one or more correctness issues. The authors claim that ZEUS has zero false negatives and a low false positive rate.

Chen et al. (2018) focus on discovering Ponzi schemes on Ethereum using a machine learning classifier built with features from user accounts and *op codes* from the smart contract bytecode. A Ponzi scheme is a classic type of fraud, similar to the pyramid scheme. Authors claim to have found more than 400 Ponzi schemes running on Ethereum. Bartoletti et al. (2017) have also written a paper on the same topic. The infestation of Ponzi schemes in Ethereum was also discussed in a Financial Times article (Kaminska, 2017). We searched for the word `pyramid` in the code file of the studied contracts and we were able to spot a clone cluster of self-admitted pyramid schemes[24].

Finally, most recently, researchers have started to build static analysis to detect bugs in smart contracts (Tikhomirov et al., 2018; Grishchenko et al., 2018). In industry, auditing companies for smart contracts have emerged, promising to ensure that newly written smart contracts are as free of bugs as possible. Some of these companies include the Solidified Team[25] and Securify[26].

Others. Zheng et al. (2018) propose high-level and low-level performance metrics for different blockchain systems (e.g., Ethereum), including a real-time monitoring framework. The authors claim that the monitoring framework has much lower overhead and offers richer performance information compared to prior approaches.

9 Threats to Validity and Limitations

Construct Validity. We detect clones using Deckard. Instead of arbitrarily defining configuration parameters, we performed a careful sensitivity analysis (Section 5). Our sensitivity analysis relied on 10 known clones (control group). Choosing a different control group would possibly lead to the selection of a different *similarity* threshold, which would in turn influence the set of clone clusters detected by the tool. Using other parameter configurations, as well as other clone detectors, is a future work endeavor.

With regards to the definition of clone types, we relied on the taxonomy of Bellon et al. (2007). For type-2 clones in particular, we relied on the notion of parameterized clones, which was proposed in the seminal paper by Baker (1992) and reused by studies in the field (Bettenburg et al., 2012; Kamiya et al., 2002; Wahler et al., 2004). In fact, some studies embed the parameterization notion in the type-2 clone definition (Roy et al., 2009).

[24] A pyramid scheme contract: `https://etherscan.io/address/0x09f55c2d116a5833d41ba9208216d11a7cdba4b3#code`

[25] `https://www.youtube.com/channel/UCpEUyenjL908MFMCO-J_yhw`

[26] `https://securify.ch`

In the authorship analysis described as part of RQ2 (Section 6.2.3), we determine the author of a smart contract by observing the blockchain address that created it. However, we cannot guarantee that each address corresponds to a different developer, since a developer could use multiple accounts (Section A.2). We believe, nevertheless, that this scenario is not the norm. Furthermore, in the authorship analysis, we manually investigated extreme cases where a cluster had several contracts developed by few developers. Such an analysis is not biased in case developers use multiple accounts.

External Validity. More generally, several approaches and implementations have been proposed to detect code clones (Roy and Cordy, 2007; Koschke, 2008; Sheneamer and Kalita, 2016). Our paper uses Deckard and consists of a first step towards a deeper understanding of code clones in blockchain-based platforms.

The population investigated in this study consisted of all verified smart contracts available on Etherscan at the time of data collection. Therefore, our results may not generalize to all smart contracts in Ethereum. However, the goal of this paper is not to build a theory that applies to all contracts, but rather to make developers and researchers aware that cloning is the *modus-operandi* of developing smart contracts. Nevertheless, additional replication studies are required in order to generalize our results to non-verified contracts as well as smart contracts deployed in other blockchain platforms, such as EOS and POA.

Limitations. Solidity is a new programming language compared to others like C#, C++, and Java. Therefore, the availability of robust parsers is more constrained. Our study relies on a Solidity parser that is still under development and therefore subject to bugs. However, we could not find any problem in the produced ASTs. As part of this study, we contributed to the parser's development by opening issues in its GitHub repository and discussing them with the developers.

10 Conclusion

The growing number of smart contracts being deployed in the Ethereum blockchain platform has attracted the attention of media outlets, industries, and researchers. In this paper, we investigated code cloning in verified contracts from Ethereum. We focused on three key aspects, namely: defining the prevalence of cloned code and categorizing the types of clones (Section 6.1), understanding the characteristics of clone clusters (Section 6.2), and determining whether smart contracts contain code blocks (subcontracts, libraries, and interfaces) that are identical to those published by the OpenZeppelin project (Section 6.3). To the best of our knowledge, this is the first study to investigate code cloning at the source code level in Ethereum.

We observed that *developers frequently clone contracts*. In particular, only 20.8% of the studied contracts are not a clone of any other contract. Also,

43.3% of the studied contracts are type-2 clones, suggesting that developers rely on templates to develop smart contracts. With regards to the characteristics, we observe that: (i) 9 out of the top-10 largest clone clusters are token managers, (ii) most of the activity of a cluster tends to be concentrated on a few contracts, and (iii) contracts in a cluster to be created by several authors. Finally, we note that the studied contracts have different ratios of code blocks that are identical to those provided by the OpenZeppelin project.

The main take-away message from this paper is that cloning is a common practice in Ethereum. Due to the immutability of smart contracts, as well as the impossibility of reverting transactions once they are deemed final, the noteworthy prevalence of clones yields direct implications to the security, development, and usage of smart contracts. More generally, despite the hype around blockchain technology, researchers should be aware that there is little diversity among verified smart contracts. Finally, by providing a supplementary material package with the data produced as part of this study, we encourage other researchers to build upon our work and gain a deeper understanding of cloning in Ethereum.

Acknowledgements This research has been supported by the Natural Sciences and Engineering Research Council (NSERC), as well as JSPS KAKENHI Japan (Grant Numbers: JP16K12415 and JP19J23477). This study leveraged the computational resources provided by the Microsoft Azure for Research program.

References

Baker BS (1992) A program for identifying duplicated code. In: Computer Science and Statistics: Proceedings of the 24th Symposium on the Interface, vol 24, pp 49–57

Bartoletti M, Carta S, Cimoli T, Saia R (2017) Dissecting ponzi schemes on ethereum: identification, analysis, and impact. CoRR abs/1703.03779, URL http://arxiv.org/abs/1703.03779, 1703.03779

Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E (2007) Comparison and evaluation of clone detection tools. IEEE Trans Softw Eng 33(9):577–591, DOI 10.1109/TSE.2007.70725, URL http://dx.doi.org/10.1109/TSE.2007.70725

Bettenburg N, Shang W, Ibrahim WM, Adams B, Zou Y, Hassan AE (2012) An empirical study on inconsistent changes to code clones at the release level. Sci Comput Program 77(6):760–776, DOI 10.1016/j.scico.2010.11.010, URL http://dx.doi.org/10.1016/j.scico.2010.11.010

Bojanowski P, Grave E, Joulin A, Mikolov T (2017) Enriching word vectors with subword information. Transactions of the Association for Computational Linguistics 5:135–146

Ceriani L, Verme P (2012) The origins of the gini index: extracts from variabilità e mutabilità (1912) by corrado gini. The Journal of Economic Inequality 10(3):421–443, DOI 10.1007/s10888-011-9188-x, URL https://doi.org/10.1007/s10888-011-9188-x

Chen W, Zheng Z, Cui J, Ngai E, Zheng P, Zhou Y (2018) Detecting ponzi schemes on ethereum: Towards healthier blockchain technology. In: Proceedings of the 2018 World Wide Web Conference, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, WWW '18, pp 1409–1418, DOI 10.1145/3178876.3186046, URL https://doi.org/10.1145/3178876.3186046

Cordy JR, Roy CK (2011) The nicad clone detector. In: Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension, IEEE Computer Society, USA, ICPC '11, p 219–220, DOI 10.1109/ICPC.2011.26, URL https://doi.org/10.1109/ICPC.2011.26

di Angelo M, Salzer G (2019) A survey of tools for analyzing ethereum smart contracts. In: 2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON), pp 69–78, DOI 10.1109/DAPPCON.2019.00018

Dijkstra EW (1982) On the Role of Scientific Thought, Springer New York, New York, NY, pp 60–66. DOI 10.1007/978-1-4612-5695-3_12, URL https://doi.org/10.1007/978-1-4612-5695-3_12

Economist T (2018) Blockchain technology may offer a way to re-decentralise the internet. https://www.economist.com/special-report/2018/06/30/blockchain-technology-may-offer-a-way-to-re-decentralise-the-internet, [Online; accessed 10-August-2018]

Fröwis M, Böhme R (2017) In code we trust? In: Garcia-Alfaro J, Navarro-Arribas G, Hartenstein H, Herrera-Joancomartí J (eds) Data Privacy Management, Cryptocurrencies and Blockchain Technology, Springer International Publishing, Cham, pp 357–372

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Gao Z, Jayasundara V, Jiang L, Xia X, Lo D, Grundy J (2019) Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding. In: Proceedings of the 35th International Conference on Software Maintenance and Evolution, ICSME '19

Göde N, Koschke R (2009) Incremental clone detection. In: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering, IEEE Computer Society, USA, CSMR '09, p 219–228, DOI 10.1109/CSMR.2009.20, URL https://doi.org/10.1109/CSMR.2009.20

Grishchenko I, Maffei M, Schneidewind C (2018) Foundations and tools for the static analysis of ethereum smart contracts. In: Chockler H, Weissenbacher G (eds) Computer Aided Verification, Springer International Publishing, Cham, pp 51–78

Hassan AE (2009) Predicting faults using the complexity of code changes. In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ICSE '09, pp 78–88, DOI 10.1109/ICSE.2009.5070510, URL http://dx.doi.org/10.1109/ICSE.2009.5070510

Hindle A, Barr ET, Su Z, Gabel M, Devanbu P (2012) On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '12, pp 837–847, URL http://dl.acm.org/citation.cfm?id=2337223.2337322

Horwitz J, Huang Z (2018) "CryptoKitties" clones are already popping up in China. https://qz.com/1174233/cryptokitties-clones-are-already-popping-up-in-china, [Online; accessed 02-December-2019]

Jakobsson M, Juels A (1999) Proofs of work and bread pudding protocols. In: Proceedings of the IFIP TC6/TC11 Joint Working Conference on Secure Information Networks: Communications and Multimedia Security, Kluwer, B.V., Deventer, The Netherlands, The Netherlands, CMS '99, pp 258–272, URL http://dl.acm.org/citation.cfm?id=647800.757199

Jiang L, Misherghi G, Su Z, Glondu S (2007a) Deckard: Scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ICSE '07, pp 96–105, DOI 10.1109/ICSE.2007.30, URL https://doi.org/10.1109/ICSE.2007.30

Jiang L, Su Z, Chiu E (2007b) Context-based detection of clone-related bugs. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ACM, New York, NY, USA, ESEC-FSE '07, pp 55–64, DOI 10.1145/1287624.1287634, URL http://doi.acm.org/10.1145/1287624.1287634

Juergens E, Deissenboeck F, Hummel B, Wagner S (2009) Do code clones matter? In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ICSE '09, pp 485–495, DOI 10.1109/ICSE.2009.5070547, URL http://dx.doi.org/10.1109/ICSE.2009.5070547

Kalra S, Goel S, Dhawan M, Sharma S (2018) ZEUS: analyzing safety of smart contracts. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018, The Internet Society, NDSS '18

Kaminska I (2017) It's not just a Ponzi, it's a 'smart' Ponzi. https://ftalphaville.ft.com/2017/06/01/2189634/its-not-just-a-ponzi-its-a-smart-ponzi/, [Online; accessed 26-August-2018]

Kamiya T, Kusumoto S, Inoue K (2002) Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans Softw Eng 28(7):654–670, DOI 10.1109/TSE.2002.1019480, URL https://doi.org/10.1109/TSE.2002.1019480

Kapser CJ, Godfrey MW (2008) "cloning considered harmful" considered harmful: patterns of cloning in software. Empirical Software Engineering 13(6):645, DOI 10.1007/s10664-008-9076-6, URL https://doi.org/10.1007/s10664-008-9076-6

Kim M, Sazawal V, Notkin D, Murphy G (2005) An empirical study of code clone genealogies. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International

Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE-13, pp 187–196, DOI 10.1145/1081706.1081737, URL http://doi.acm.org/10.1145/1081706.1081737

Koschke R (2008) Identifying and removing software clones. In: Mens T, Demeyer S (eds) Software Evolution, 1st edn, Springer, chap 2, DOI 10.1007/978-3-540-76440-3

Liu H, Yang Z, Liu C, Jiang Y, Zhao W, Sun J (2018) Eclone: Detect semantic clones in ethereum via symbolic transaction sketch. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE 2018, pp 900–903, DOI 10.1145/3236024.3264596, URL http://doi.acm.org/10.1145/3236024.3264596

Liu H, Yang Z, Jiang Y, Zhao W, Sun J (2019) Enabling clone detection for ethereum via smart contract birthmarks. In: Proceedings of the 27th International Conference on Program Comprehension, IEEE Press, Piscataway, NJ, USA, ICPC '19, pp 105–115, DOI 10.1109/ICPC.2019.00024, URL https://doi.org/10.1109/ICPC.2019.00024

Lopes CV, Maj P, Martins P, Saini V, Yang D, Zitny J, Sajnani H, Vitek J (2017) Déjàvu: A map of code duplicates on github. Proc ACM Program Lang 1(OOPSLA):84:1–84:28, DOI 10.1145/3133908, URL http://doi.acm.org/10.1145/3133908

Luu L, Chu DH, Olickel H, Saxena P, Hobor A (2016) Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ACM, New York, NY, USA, CCS '16, pp 254–269, DOI 10.1145/2976749.2978309, URL http://doi.acm.org/10.1145/2976749.2978309

Mockus A (2007) Large-scale code reuse in open source software. In: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development, IEEE Computer Society, Washington, DC, USA, FLOSS '07, pp 7–, DOI 10.1109/FLOSS.2007.10, URL http://dx.doi.org/10.1109/FLOSS.2007.10

Popper N (2017) Understanding Ethereum, Bitcoin's Virtual Cousin. https://www.nytimes.com/2017/10/01/technology/what-is-ethereum.html, [Online; accessed 10-August-2018]

Romano J, Kromrey J, Coraggio J, Skowronek J (2006) Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys? In: Annual meeting of the Florida Association of Institutional Research, pp 1–3

Roos P (2015) Fast and precise statistical code completion. In: Proceedings of the 37th International Conference on Software Engineering - Volume 2, IEEE Press, Piscataway, NJ, USA, ICSE '15, pp 757–759, URL http://dl.acm.org/citation.cfm?id=2819009.2819158

Roy CK, Cordy JR (2007) A survey on software clone detection research. Technical Report 2007-541, School of Computing - Queen's University

Roy CK, Cordy JR, Koschke R (2009) Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Sci Comput

Program 74(7):470–495, DOI 10.1016/j.scico.2009.02.007, URL http://dx.doi.org/10.1016/j.scico.2009.02.007

Sajnani H, Saini V, Svajlenko J, Roy CK, Lopes CV (2016) Sourcerercc: Scaling code clone detection to big-code. In: Proceedings of the 38th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '16, p 1157–1168, DOI 10.1145/2884781.2884877, URL https://doi.org/10.1145/2884781.2884877

Shannon CE, Weaver W (1963) A Mathematical Theory of Communication. University of Illinois Press, Champaign, IL, USA

Sheneamer A, Kalita J (2016) A survey of software clone detection techniques. International Journal of Computer Applications 137(10):1–21, published by Foundation of Computer Science (FCS), NY, USA

Skvorc B (2018) 15 Alternatives to CryptoKitties You Had No Idea Existed. https://qz.com/1174233/cryptokitties-clones-are-already-popping-up-in-china, [Online; accessed 02-December-2019]

Swan M (2015) Blockchain: Blueprint for a New Economy, 1st edn. O'Reilly Media, Inc.

Szabo N (1994) Smart Contracts. http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html, [Online; accessed 26-August-2018]

Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y (2018) Smartcheck: Static analysis of ethereum smart contracts. In: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, ACM, New York, NY, USA, WETSEB '18, pp 9–16, DOI 10.1145/3194113.3194115, URL http://doi.acm.org/10.1145/3194113.3194115

Ukkonen E (1992) Approximate string-matching with q-grams and maximal matches. Theoretical Computer Science 92(1):191 – 211, DOI https://doi.org/10.1016/0304-3975(92)90143-4, URL http://www.sciencedirect.com/science/article/pii/0304397592901434

Wahler V, Seipel D, Gudenberg JWv, Fischer G (2004) Clone detection in source code by frequent itemset techniques. In: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop, IEEE Computer Society, Washington, DC, USA, SCAM '04, pp 128–135, DOI 10.1109/SCAM.2004.5, URL https://doi.org/10.1109/SCAM.2004.5

Wood G (2017) Ethereum: A Secure Decentralised Generalised Transaction Ledger - EIP-150 Revision. http://yellowpaper.io/, [Online; accessed 10-August-2018]

Zheng P, Zheng Z, Luo X, Chen X, Liu X (2018) A detailed and real-time performance monitoring framework for blockchain systems. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ACM, New York, NY, USA, ICSE-SEIP '18, pp 134–143, DOI 10.1145/3183519.3183546, URL http://doi.acm.org/10.1145/3183519.3183546

Appendix

A Background

In this section, we describe concepts that are key to our study. Sections A.1 defines blockchain. Section A.2 describes Ethereum accounts. Section A.3 introduces *smart contracts*, including how one deploys, verifies, and executes smart contracts. Finally, Section A.4 defines *token*, *token contracts*, and *mintable token contracts*.

A.1 Blockchain

A blockchain is a distributed, chronological database of transactions that is shared and maintained across nodes that participate in a peer-to-peer network. Ethereum and Bitcoin are two of the most popular blockchain platforms. As of January 2020, the Ethereum platform holds a remarkable market capitalization of 15.67 billion USD[27].

Transactions are at the heart of blockchains. The name *blockchain* comes from the manner in which transactions are stored. More specifically, transactions are packaged into blocks that are linked to one another as a chain. Adding a new transaction to a blockchain requires confirmation from several nodes of the network, which all abide to a certain consensus protocol. Such a protocol is designed to be costly (e.g., in terms of computing power or time) in order to ensure that tampering with the data is infeasible. The Ethereum platform uses the computationally costly *Proof-of-Work (PoW)* consensus protocol (Jakobsson and Juels, 1999), which requires nodes to solve a hard mathematical puzzle. The PoW consensus protocol ensures that there is no better strategy to find the solution to the mathematical puzzle than enumerating the possibilities (i.e., brute force). On the other hand, verification of a solution is trivial and cheap. Ultimately, the PoW consensus protocol ensures that a trustworthy third-party (e.g., a bank) is *not* needed in order to validate transactions, enabling entities who do not know or trust each other to build a dependable transaction ledger.

Once a block is appended to the blockchain, its contents cannot be altered without changing every other block that came after it. In practice, a transaction is deemed final and irreversible after six block confirmations (i.e., after six new blocks have been added to blockchain). More generally, due to the PoW consensus protocol, it is impossible to change the contents of old blocks without owning more than 50% of the computing power that runs Ethereum.

[27] *Market capitalization* is the multiplication of a company's shares by its current stock price. In the virtual coin world, a company's share corresponds to the total value of its coin supply. As of January 07th 2020, Ethereum has a total ether supply of 109,174,249, with a market price of 143.55 USD per ether, yielding a market capitalization of 15.67 billion dollars.

A.2 Ethereum Accounts

The Ethereum platform supports two types of accounts: *user accounts* and *smart contract accounts*. A user account is very simple in structure. A user account has an address (40-digit hexadecimal ID), a transaction count, and the ETH balance (ETH is the official Ethereum cryptocurrency). A contract account, in turn, holds the bytecode of a smart contract in addition to the previously mentioned fields. By means of a transaction, a user account can transfer ETH to another account, deploy a smart contract (Section A.3.2), or execute a function of a smart contract (Section A.3.4).

A.3 Smart Contracts

The key difference between Ethereum and Bitcoin is that the former supports *smart contracts*. The term *smart contract* was coined by Szabo (1994). According to him, "a smart contract is a computerized transaction protocol that executes the terms of a contract." More recently, with the advent of Ethereum and other sophisticated blockchain platforms, the concept of smart contracts has become much broader, representing any general-purpose computation. For instance, smart contracts have been used to implement crowdfunding campaigns (e.g., by selling tokens to the public, similarly to an IPO[28]), RPG games (e.g., MyCryptoHeroes[29]), and (crypto)currency trading platforms (e.g., IDEX[30]). Blockchain platforms that support smart contracts are known as *programmable blockchains*.

A.3.1 Source Code

The source code of a smart contract is written in the Solidity language, whose syntax is similar to that of Java. An illustrative example is shown in Figure 24. In order to enable the separation of concerns (Dijkstra, 1982), the Solidity language provides three key constructs: *subcontracts*, *libraries*, and *interfaces*. When convenient, we indistinctly refer to them as *code blocks*.

Subcontracts. Subcontracts are similar to classes (as in object-oriented programming). As such, subcontracts typically implement a certain concept (lines 27-51 and 53-65 from the example). Similarly to Java, a subcontract is deemed as abstract when at least one of their functions lacks an implementation. Abstract subcontracts cannot be instantiated, since they are meant to be used as *base* subcontracts. If a subcontract A *inherits* from a base subcontract B, then we say that A is a *child* of B (and that B is a *parent* of A).

Interfaces. The concept of interfaces comes straight from object-oriented programming. Interfaces are thus similar to abstract subcontracts, but they can-

[28] `https://en.wikipedia.org/wiki/Initial_public_offering`
[29] `https://www.mycryptoheroes.net`
[30] `https://idex.market`

```
1
2  pragma solidity ^0.4.24;                                    Compiler Version
3
4  library SafeMath {                                          Library
5
6    /**
7     * @dev Multiplies two numbers, reverts on overflow.     Code comment
8     */
9    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
10     if (a == 0) {
11       return 0;
12     }
13   uint256 c = a * b;
14   require(c / a == b);
15   return c;
16   }
17   ...
18 }
19
20 interface IERC20 {                                           Interface declaration
21
22   function totalSupply() external view returns (uint256);
23   function balanceOf(address who) external view returns (uint256);
24   ...                                                        Interface functions
25 }
26 Subcontract   Interface realization
27 contract ERC20 is IERC20 {
28
29   using SafeMath for uint256;
30
31   mapping (address => uint256) private _balances;
32   mapping (address => mapping (address => uint256)) private _allowed;
33   uint256 private _totalSupply;
34
35   /**
36    * @dev Total number of tokens in existence
37    */  Function name                  Return type
38   function totalSupply() public view returns (uint256) {
39     return _totalSupply;              Function modifiers
40   }
41
42   /**
43    * @dev Gets the balance of the specified address.
44    * @param owner The address to query the balance of.
45    * @return An uint256 with the amount owned by the passed address.
46    */
47   function balanceOf(address owner) public view returns (uint256) {
48     return _balances[owner];
49   }
50   ...
51 }
52
53 contract MyCoin is ERC20 { Subcontract inheritance
54   string public symbol;
55   string public name;
56   uint8 public decimals;
57
58   function MyCoin() public { Subcontract constructor
59     symbol = "MC";
60     name = "MyCoin";
61     decimals = 18;
62   ...
63   }
64   ...
65 }
```

Fig. 24: An example of a smart contract written in Solidity.

not have any implemented functions (lines 20-25 from the example). In Solidity, subcontracts *realize* an interface by inheriting from it (line 27 from the example).

Libraries. A library is an isolated piece of code that is meant to be stateless. Libraries often provide a set of utility methods that are mindful of corner-cases or that optimize processing time. For instance, a developer might implement a library that performs mathematical operations without overflows exceptions (lines 4-18 from the example).

A.3.2 Deployment

In Ethereum, a user account can *deploy* smart contracts. The deployment of a smart contract is done by means of a transaction[31] that is sent to the blockchain. Such a transaction is commonly referred to as the *contract creation transaction*. Upon the successful execution of this transaction, the contract is deployed in the blockchain and receives an address. This transaction also records the address of the user account that deployed the contract. This user account is often referred to as the *creator* (*author*) of the contract.

A.3.3 Verification

When a user account deploys a smart contract to the Ethereum platform, only the bytecode is stored in the blockchain. Therefore, it is up to the developer to publish the source code of the smart contract. The Etherscan website, which is the primary Ethereum dashboard website, provides a code transparency mechanism known as *contract verification*. This mechanism offers developers the possibility of publishing the source code of a smart contract on Etherscan, so it becomes available to anyone that is interested in the Ethereum platform. The code verification mechanism works as follows: (i) the developer uploads a *flattened*[32] version of the source code (i.e., a single file containing all the source code) and indicates a particular version of the Solidity compiler, (ii) Etherscan compiles the code using the developer-indicated compiler version, (iii) Etherscan checks if the generated bytecode matches the bytecode that is stored in the blockchain. If there is a perfect match, then the smart contract is deemed as *verified* and the flattened version of the source code becomes publicly available on Etherscan. We refer to this flattened version of the source code that is published on Etherscan as the *code file* of a verified contract. A list of verified contracts can be found at `https://etherscan.io/contractsVerified`.

[31] Example of a transaction that created a smart contract: `https://etherscan.io/tx/0xebcbe706f9959c8b98a72bcd42fed545d3cf60fe3fa801186d5fef2249dac91a`

[32] There are tools to help developers flatten Solidity code. An example is `truffle-flattener`, available at `https://www.npmjs.com/package/truffle-flattener`.

A.3.4 Execution

In Ethereum, a user account can not only deploy but also *execute* contracts. A user account executes a smart contract by sending transactions to it. These transactions carry data that specify which function should be executed, as well as data regarding the input parameters of this function. Figure 25 shows an example of a transaction in which the transaction issuer (a user account) transferred tokens to another user account by executing the `transfer(address _to, uint256 _value)` function of a smart contract.

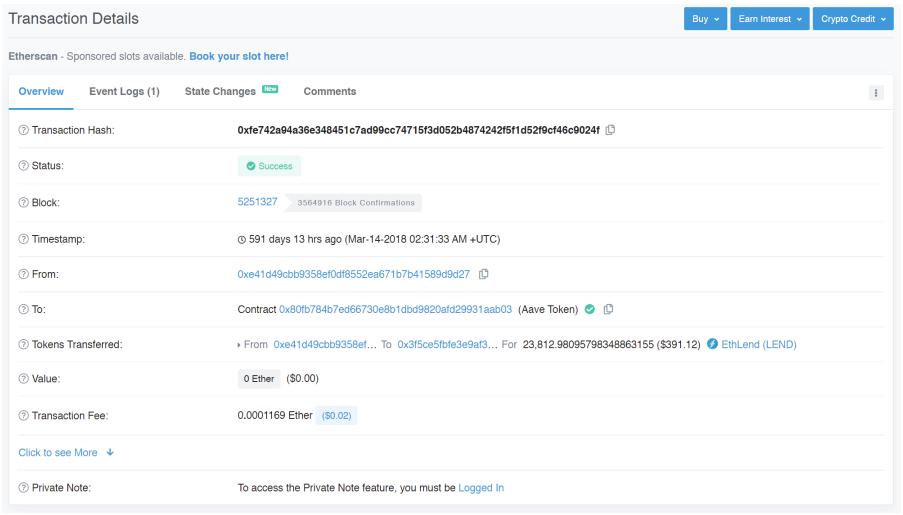

Fig. 25: An example of a smart contract transaction. Image extracted from Etherscan[33].

A.4 Cryptocurrency, tokens, and coins

A *cryptocurrency* is a virtual artifact which represents money. A cryptocurrency is native to its own blockchain. In the case of Ethereum, the cryptocurrency is called Ether and is abbreviated as ETH. Ether can be transferred between user accounts. Ether (ETH) is not much different than traditional currencies like USD Dollars (USD) and Euros (EUR). The only practical difference is that we have metal coins and pieces of paper to represent dollars and euros in the physical world. Instead, cryptocurrencies are purely virtual.

Tokens are created on top of existing blockchains. Tokens are used to represent digital assets that are tradeable (and usually fungible), including ev-

[33] `https://etherscan.io/tx/0xfe742a94a36e348451c7ad99cc74715f3d052b4874242f5f1d52f9cf46c9024f`.

erything from commodities to voting rights. Every token has a name and an acronym (popularly known as a *symbol*) and any smart contract can define a new token. It is common for tokens to represent money. Therefore, in practice, (crypto)coins and tokens are frequently used interchangeably. For instance, a crowdfunding initiative that is implemented as a distribution of tokens is more commonly referred to as an ICO (Initial *Coin* Offering) instead of an ITO (Initial *Token* Offering). There are several physical and virtual currency exchanges (e.g., IDEX[34]) around the world that buy and sell (crypto)coins, as well as exchange one (crypto)coin for another.

A.4.1 Token Contract

A *token contract* is a special kind of smart contract that defines a token and keeps track of its balance across user accounts. Ethereum has two main technical standards for the implementation of tokens, known as the *ERC20* and *ERC721*. The standardization allows contracts to operate on different tokens seamlessly, thus fostering interoperability between smart contracts. From an implementation perspective, ERC20 and ERC721 are object-oriented interfaces defining several functions, such as `totalSupply()`, `balanceOf(address who)`, and `transfer(address to, uint256 value)` (check IERC20 in Figure 24).

A.4.2 Mintable Token and Mintable Token Contact

A *mintable token* is a special kind of token that has a *non-fixed total supply*. Most *mintable token contracts* are ERC20 token contracts with an added `mint()` function, which increases the total token supply upon invocation. Optionally, a `burn()` function is also included to decrease the total supply. Bitcoin (BTC), the official cryptocurrency of the homonymous blockchain plaform, is a mintable token. In particular, 12.5 newly created BTCs are given as a reward to those who put the next block on the Bitcoin blockchain platform. Ether (ETH) is *not* mintable.

B Additional Resources

B.1 Top-10 largest clusters: UML diagram of each representative contract

[34] `https://idex.market`

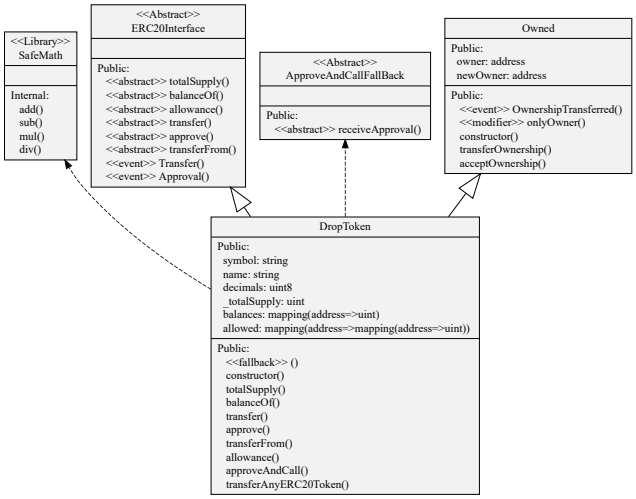

Fig. 26: Representative contract of clone cluster 1. This contract is deployed at the address `0x4672bAD527107471cB5067a887f4656D585a8A31`.

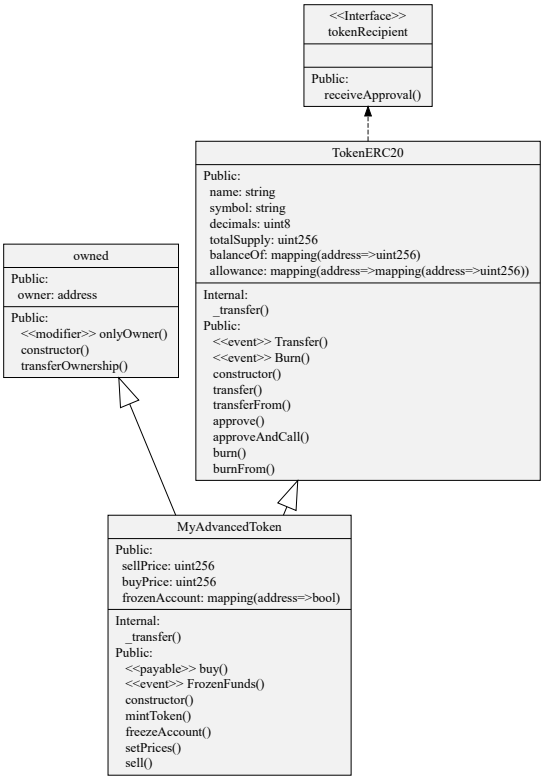

Fig. 27: Representative contract of clone cluster 2. This contract is deployed at the address `0x8912358d977e123b51ecad1ffa0cc4a7e32ff774`.

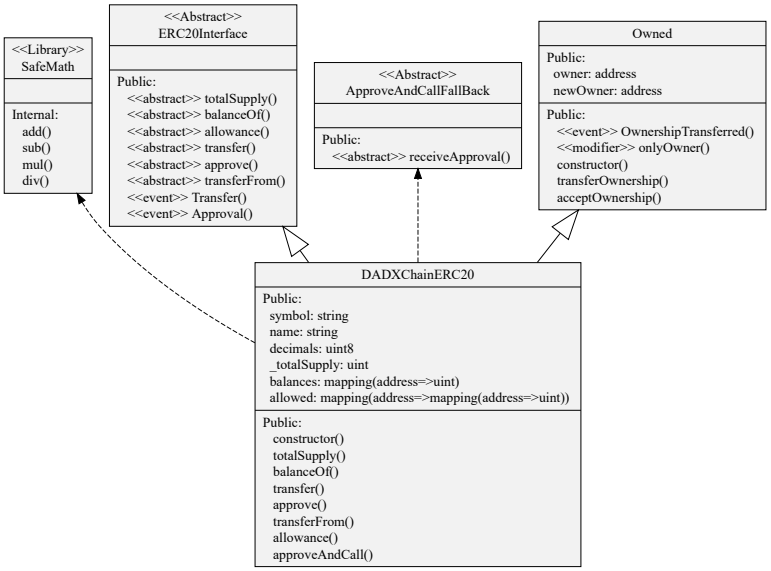

Fig. 28: Representative contract of clone cluster 3. This contract is deployed at the address `0x30392c252da07b69194972e9f770b6dd5deb7af8`.

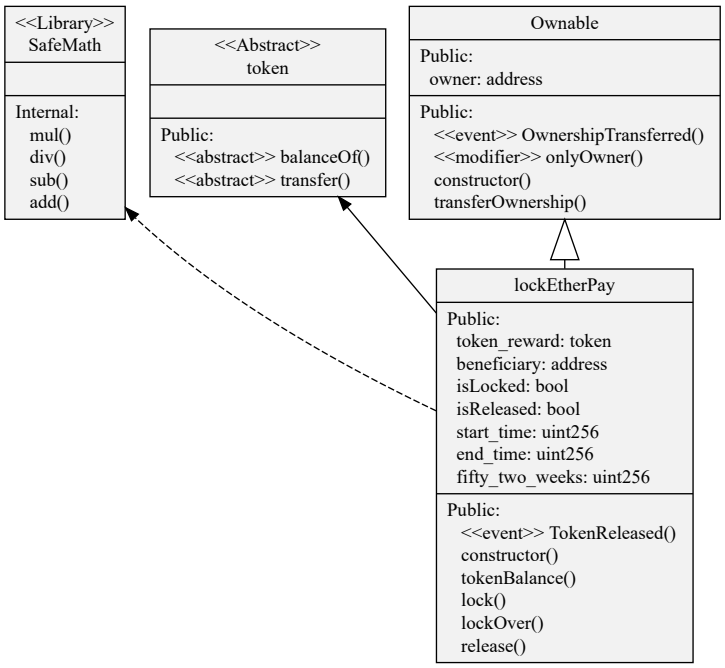

Fig. 29: Representative contract of clone cluster 4. This contract is deployed at the address `0xa7d7609766d7fcfaf38eda123454bf94b1c1abf0`.

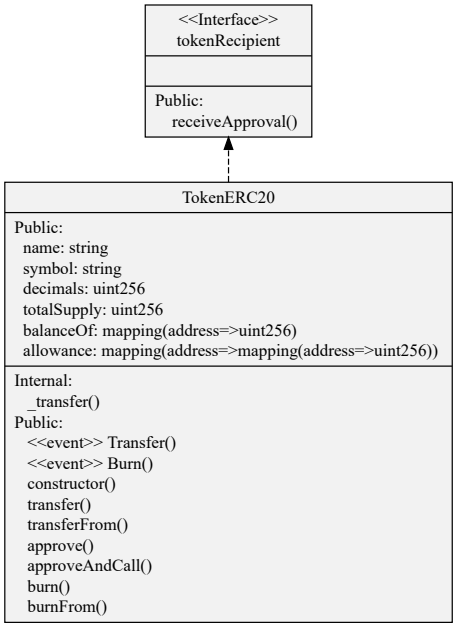

Fig. 30: Representative contract of clone cluster 5. This contract is deployed at the address `0x9064c91e51d7021a85ad96817e1432abf6624470`.

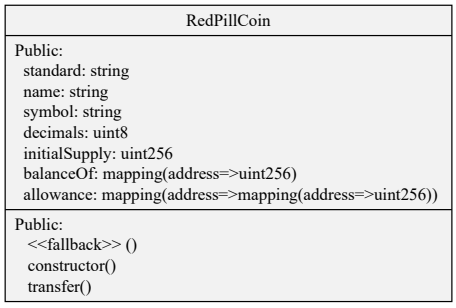

Fig. 31: Representative contract of clone cluster 6. This contract is deployed at the address `0x47a892bf7336a120ee69b2db6acb552acad5f46d`.

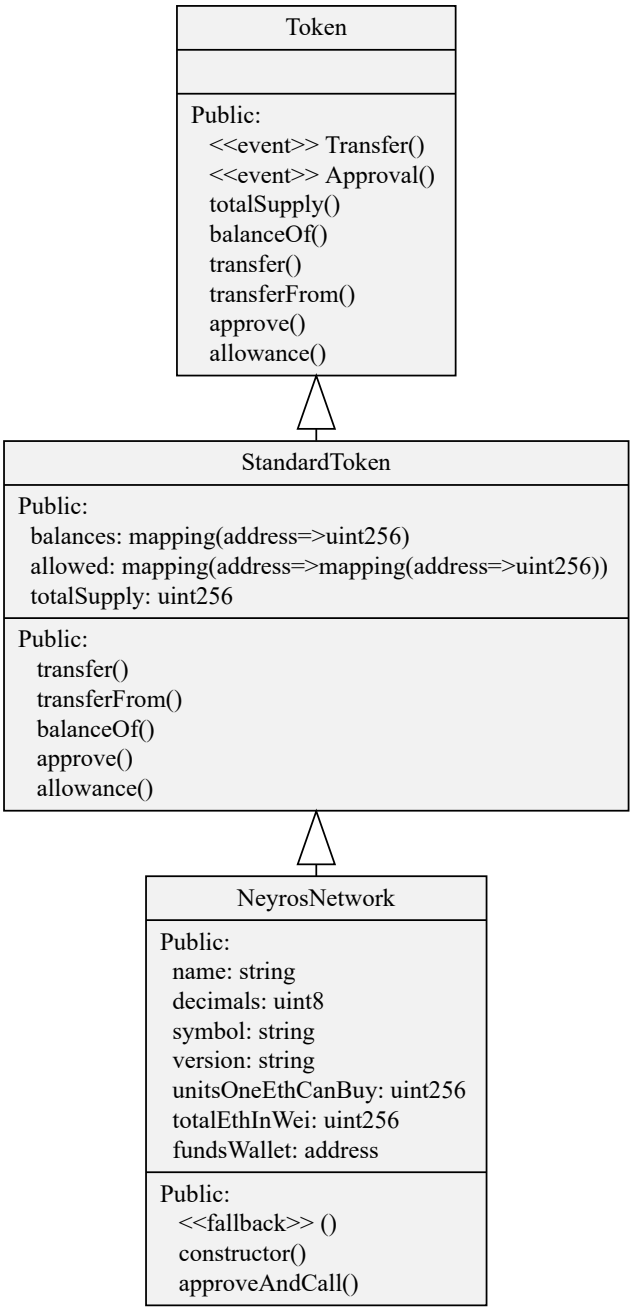

Fig. 32: Representative contract of clone cluster 7. This contract is deployed at the address `0x1d8e5dcf365864fcda8ab39d1f60d66ee2b82770`.

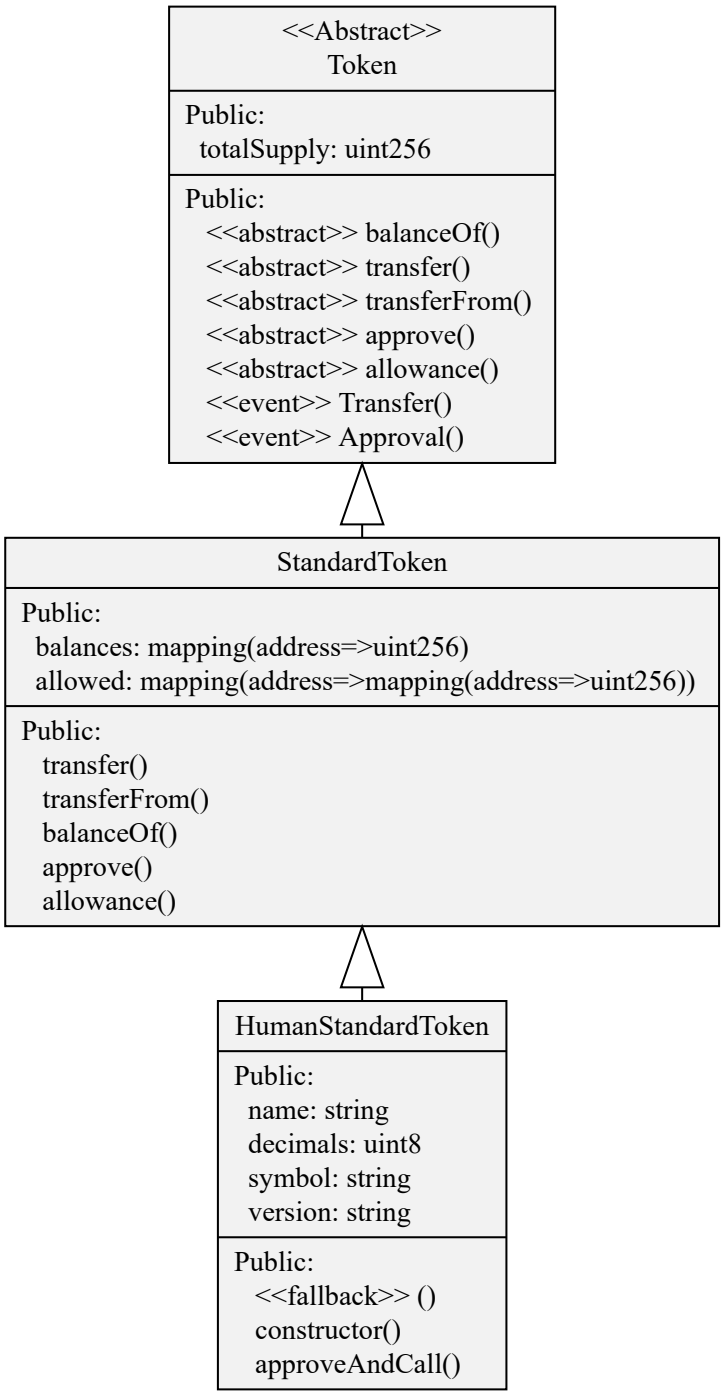

Fig. 33: Representative contract of clone cluster 8. This contract is deployed at the address `0x9a642d6b3368ddc662ca244badf32cda716005bc`.

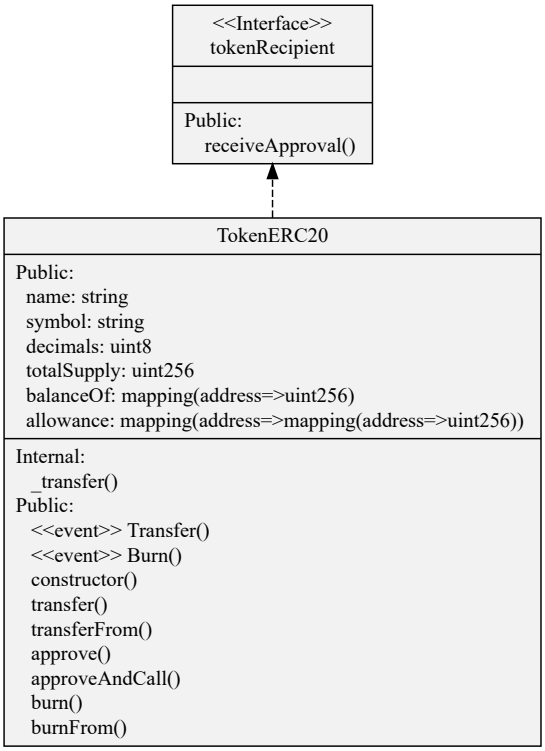

Fig. 34: Representative contract of clone cluster 9. This contract is deployed at the address `0x2eb86e8fc520e0f6bb5d9af08f924fe70558ab89`.

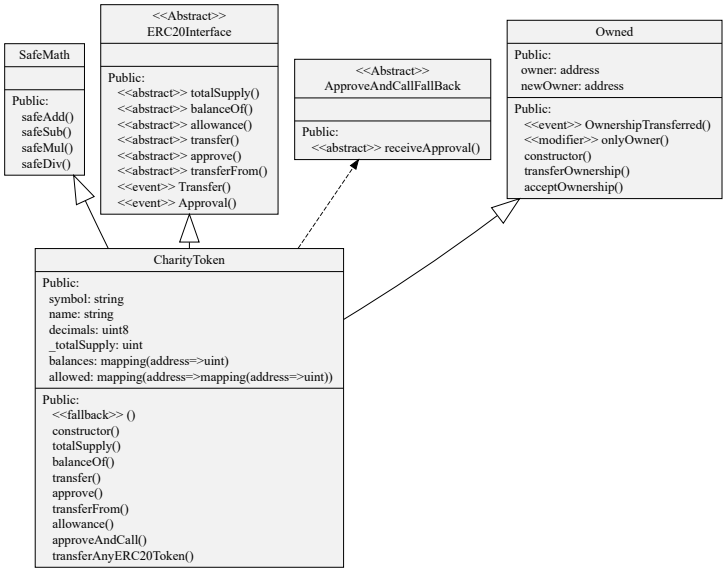

Fig. 35: Representative contract of clone cluster 10. This contract is deployed at the address `0x0a2eaa1101bfec3844d9f79dd4e5b2f2d5b1fd4d`.