Characterizing Code Clones in the Ethereum Smart Contract Ecosystem

Ningyu He[1], Lei Wu[2], Haoyu Wang[3]✉, Yao Guo[1]✉, and Xuxian Jiang[4]

[1] MOE Key Lab of HCST, School of EECS, Peking University, China
[2] School of Cyber Science and Technology, Zhejiang University, China
[3] Beijing University of Posts and Telecommunications, China
[4] PeckShield, Inc.

Abstract. In this paper, we present the first large-scale and systematic study to characterize the code reuse practice in the Ethereum smart contract ecosystem. We first performed a detailed similarity comparison study on a dataset of 10 million contracts we had harvested, and then we further conducted a qualitative analysis to characterize the diversity of the ecosystem, understand the correlation between code reuse and vulnerabilities, and detect the plagiarized DApps. Our analysis revealed that over 96% of the contracts had duplicates, while a large number of them were similar, which suggests that the ecosystem is highly homogeneous. Our results also suggested that roughly 9.7% of the similar contract pairs have exactly the same vulnerabilities, which we assume were introduced by code clones. In addition, we identified 41 DApps clusters, involving 73 plagiarized DApps which had caused huge financial loss to the original creators, accounting for 1/3 of the original market volume.

Keywords: Code Clone · Smart Contract · Ethereum · Vulnerability.

1 INTRODUCTION

Smart contracts, the most important innovation of Ethereum, provide the ability to "digitally facilitate, verify, and enforce the negotiation or performance of a contract" [46], while the correctness of its execution is ensured by the consensus protocol of Ethereum. Such a courageous attempt has been approved by the market, *i.e.*, Ethereum's market cap was around $14.5B on February 26th, 2019 [2], the largest volume besides Bitcoin. As of this writing, roughly 10 million smart contracts have been deployed on the Ethereum Mainnet.

Smart contracts are typically written in higher level languages, *e.g.*, Solidity [39] (a language similar to JavaScript and C++), then compiled to Ethereum Virtual Machine (EVM) bytecode. As one of the most important rules on Ethereum, "Code is Law", means all executions and transactions are final and immutable.

As a result, one main characteristic of smart contracts is that a considerable number of them published source code to gain the users' trust and prove the security of their code, especially for the popular ones [22]. This feature is more noticeable for Decentralized Applications (DApps for short, which consist of one or more contracts). In general, their code base should be available for scrutiny and governed by autonomy, distinguished from the traditional closed-source applications that require the end users to trust the developers in terms of decentralization as they cannot directly access data via any central source.

```
DiviesInterface constant private Divies = \
    DiviesInterface(0xc7029...);
JIincForwarderInterface constant private Jekyll_Island_Inc = \
    JIincForwarderInterface(0xdd49...);
PlayerBookInterface constant private PlayerBook = \
    PlayerBookInterface(0xD60d...);
F3DexternalSettingsInterface constant private extSettings = \
    F3DexternalSettingsInterface(0x3296...);

string constant public name = "FoMo3D Long Official";
string constant public symbol = "F3D";
uint256 private rndExtra_ = extSettings.getLongExtra();
uint256 private rndGap_ = extSettings.getLongGap();
uint256 constant private rndInit_ = 1 hours;
uint256 constant private rndInc_ = 30 seconds;
uint256 constant private rndMax_ = 24 hours;
```

(a) Original Fomo3D

```
PlayerBookInterface constant private PlayerBook = \
    PlayerBookInterface(0x8676...);

string constant public name = "imfomo Long Official";
string constant public symbol = "imfomo";
uint256 private rndExtra_ = 30;
uint256 private rndGap_ = 30;
uint256 constant private rndInit_ = 10 minutes;
uint256 constant private rndInc_ = 60 seconds;
uint256 constant private rndMax_ = 10 minutes;
```

(b) Modified Fomo3D

Fig. 1. The original Fomo3D and a plagiarized contract from it.

However, the open-source nature of smart contracts has provided convenience for plagiarists to create **contract clones**, *i.e.*, copying code from other available contracts. The impact of contract clones is two-fold. On one hand, the plagiarists could insert arbitrary/malicious code into the normal contracts. A typical example is the so-called `honeypot smart contracts` [26–28], which are scam contracts that try to fool users with stealthy tricks. On the other hand, as many smart contracts are suffering from serious vulnerabilities, the copy-paste vulnerabilities would be inherited by the plagiarized contracts.

Here, we use Fomo3D [36] as a motivating example, which is a popular and phenomenal Ponzi-like game. At its peak in 2018, Fomo3D had over $10,000$ daily active users with a volume of over $40,000$ ETHs [35]. As a result, numerous Fomo3D-like games sprang up with plagiarism behaviors by simply reusing the source code of the original one. Unfortunately, some hackers had figured out the design flaw of the airdrop mechanism in the original Fomo3D [25]. Consequently, almost all the awkward imitators were exposed to those attackers. *LastWinner*, one of the most successful followers of Fomo3D, was attacked and lost more than $5,000$ ETHs within 4 days [31]. Figure 1 shows a plagiarized contract example originated from Fomo3D [35]. Interestingly, the vulnerable part was kept wholly intact by the plagiarist, but all the dependent contracts, and some arguments like round timer and round increment, were modified to make it appear as a brand new game as shown in Figure 1.

This paper. We present a large scale systematic study to characterize the code clone behaviors of Ethereum smart contracts in a comprehensive manner. To this end, we have collected by far the largest Ethereum smart contract dataset with nearly 10 million smart contracts deployed between July 2015 to December 2018. To address the scalability issues introduced by the large scale dataset, we first seek to identify the duplicate contracts by removing code from unrelated functions (*e.g.*, creation code and Swarm code), and tokenizing the code to keep opcodes only. After the pre-processing step to remove duplicate contracts, the dataset has been shrunk to less than 1% of the original size. For the remaining $78,611$ distinct contracts, we take advantage of a customized fuzzy hashing approach to generate the fingerprints and then conduct a pair-wise similarity comparison. Specifically, we adopt a pruning strategy to discard "very different" contracts by comparing the meta features (*e.g.*, length of opcode), to accelerate the comparison procedure. Based on a similarity threshold of 70, we are able to

identify 472,663 similar smart contract pairs (with 47,242 contracts involved) for user-created contracts, which suggested that over 63.29% of the distinct user-created contracts have at least one similar contract in our dataset.

Then, we further seek to understand the reasons leading to contract clones and characterize their security impacts.

(1) The reasons leading to contract clones. Over 60% of the distinct contracts were grouped into roughly 10K clusters, while the cluster distribution follows a typical Pareto principle. Top 20% of the clusters occupied over 60% of the distinct contracts. With regard to the whole dataset including all the duplicates, the top 1% of the clusters account for 95% of the contracts. ERC20 token contracts, ICO and AirDrop, and Game contracts are the most popular clusters. A large number of similar contracts were created based on the same template (*i.e.*, ERC20 template). We have manually summarized a list of 53 common templates used in the Ethereum smart contract ecosystem. **This result reveals the homogeneity nature of the smart contract ecosystem.**

(2) Vulnerability Provenance. Copy-paste vulnerabilities were prevalent in most popular software systems. Here, we study the relationship between contract clones and the presence of vulnerabilities, from two aspects. First, we scanned all the unique contracts using a state-of-the-art vulnerability scanner [37]. Over 20,346 distinct smart contracts (27.26%) contain at least one vulnerability. **Considering the large number of duplicates, we were able to identify ten-fold vulnerable contracts (**205,010**)**. Then, for the distinct contracts, as a number of them have similar code, we further compare whether they were exposed to similar vulnerabilities. **Overall, our results suggest that roughly 9.7% of the similar contract pairs have exactly the same vulnerabilities, which we assume were introduced by code clones.**

(3) Plagiarized DApps. As a DApp is more complicated than a smart contract, *i.e.*, one DApp could include one or more contracts, thus we further study the similarity between DApps, seeking to identify the plagiarized ones. Using a bipartite graph matching approach, we identified 41 DApp clusters, involving 73 plagiarized DApps. **The plagiarized ones have caused huge financial loss to the original creators, accounting for 1/3 of the original volume.**

To the best of our knowledge, this is the first systematic study of code clones in the Ethereum smart contract ecosystem *at scale*. Our results revealed the highly homogeneous nature of the ecosystem, *i.e.*, code clones are prevalent, which helps spread vulnerabilities and makes it easier for plagiarists. Our results motivated the need for research efforts to identify security issues introduced by copy-paste behaviors. Our efforts can positively contribute to the smart contract ecosystem, and promote the best operational practices for developers.

2 BACKGROUND

2.1 Ethereum

External Owned Account vs. Contract Account. The basic unit of Ethereum is an *account* and there are two types of accounts [21]: External Owned Account (EOA) and Contract Account. An EOA is controlled by private keys that are

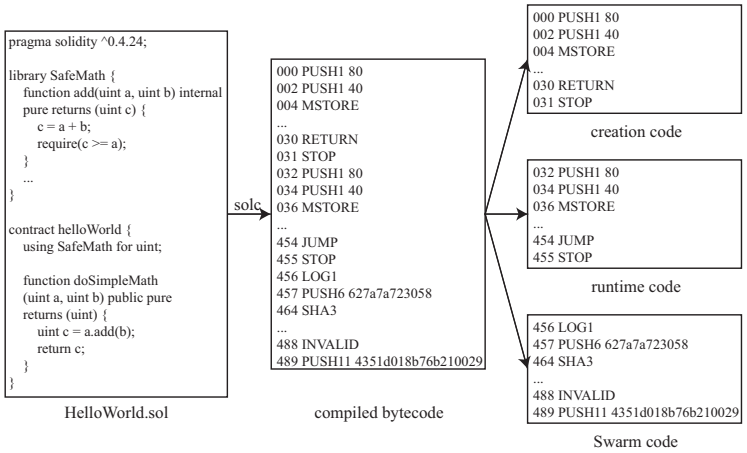


Fig. 2. An example of Helloworld.sol and its corresponding bytecode.

externally owned by a user. More importantly, there is no code associated with it. One can send messages from an EOA by creating and signing a transaction. On the contrary, a contract account is controlled by its associated contract code, which might be activated on receiving a message.

User-created Contract vs. Contract-created Contract. Smart contract can be created either by users, or by existing contracts. In this paper, we follow the terminology "user-created contract" and "contract-created contract" adopted by Kiffer et al. [60] to distinguish these two types of creations.

Decentralized applications (DApps). Ethereum aims to create an alternative protocol to build DApps [34], which are stored on and executed by the Ethereum system. Specifically, a DApp is *a contract or a collection of contracts that have an interface on the Internet, typically a website or a browser game, which could be interacted by players or users directly.* A number of websites emerged to host DApps lists [15–17].

2.2 Ethereum Virtual Machine (EVM)

EVM is the runtime environment for smart contracts. Specifically, a sandboxed virtual stack machine is embedded within each full Ethereum node, responsible for executing contract bytecode with a 256-bit register stack [20]. Its operators and operands are all pushed onto the stack indistinguishably, except for data that require persistent storage space. Therefore, all the immediate numbers and data to be used by the operation code will be *pushed* onto the stack.

Generally, developers implement their smart contracts with the Solidity language, then build the source code using the Solidity compiler, a.k.a. *solc*, to generate the EVM bytecode. A typical EVM bytecode is composed of three parts: *creation code*, *runtime code* and *swarm code*, as shown in Figure 2.

Creation code is only executed by EVM once during the transaction of the contract deployment. It determines the initial states of the smart contract being deployed and returns a copy of the runtime code. It usually end with the sequence: `PUSH 0x00, RETURN, STOP`, corresponding to `0x6000f300` (cf. Fig 2).

Runtime code is the most crucial part, including function selector, function wrapper, function body and exception handling. Based on the corresponding operations, EVM will execute runtime code accordingly. Besides, in order to label jumping destinations of the function selector, solc sorts functions by their signatures, *i.e.*, the leading 4 bytes of the SHA-3 hashes of function declarations with a well-defined format [38]. Accordingly, adding new functions or deleting existing ones will not affect the relative order of the remaining functions.

Swarm code is not served for execution purpose. Solc uses the metadata of a contract, including compiler version, source code and the located block number, to calculate the so-called Swarm hash, which can be used to query on *Swarm*, a decentralized storage system, to prove the consistency between the contract you see and the contract being deployed, namely *what you see is what you get*. As a result, re-deploying a smart contract would result in a different swarm code, even with the same creation code and runtime code. Swarm code always begins with `0xa165`, *i.e.*, `LOG1 PUSH 6`. The following six bytes are `0x627a7a723058`, whose leading four bytes can be decoded as "bzzr", the Swarm's URL scheme. Furthermore, Swarm code always ends with `0x0029`, which means the hash part length between `0xa165` and `0x0029` is 41 bytes long. In short, we are able to identify the swarm code quickly and precisely based on those hard-coded bytes.

3 Methodology

Overall Process. We summarize our approach in Figure 3. The pipeline starts with the dataset with nearly 10 million smart contracts we have collected. We first seek to remove duplicate smart contracts to reduce the computational workload in two steps: 1) removing the creation and Swarm code parts, which are not useful for analyzing; 2) removing all assigned values in assignment statements and function calls. To this end, the smart contracts were scanned for tokenization by generating token hashes, which allow us to capture subtle differences of the clones. After that, for the remaining contracts with distinct token hashes, we take advantage of a customized fuzzy hashing approach to generate the fingerprints. Lastly, we enforce a pair-wise comparison strategy with pruning to achieve scalability. The output of the whole analysis pipeline is a set of contract clone pairs with the corresponding similarity scores. Note that the output results will be further correlated with our in-depth analysis in Section 5, including contract clustering, vulnerability provenance and DApps plagiarism detection.

In this paper, we did not rely on heavy-weight methods such as comparing the control-flow graph (CFG) and program dependency graph (PDG), mainly due to two reasons. First, our approach should be scalable. Second, the simplicity of smart contracts and EVM bytecode, *i.e.*, the relatively simple logic and function invocations, makes it unnecessary to adopt those heavy approaches. To the best of our knowledge, we did not even identify smart contracts with heavy obfuscation. We evaluate the effectiveness of our approach in Section 4.

3.1 Pre-processing

The purpose of pre-processing is two-fold: first identifying the duplicate contracts, and then tokenizing contracts for further comparison.

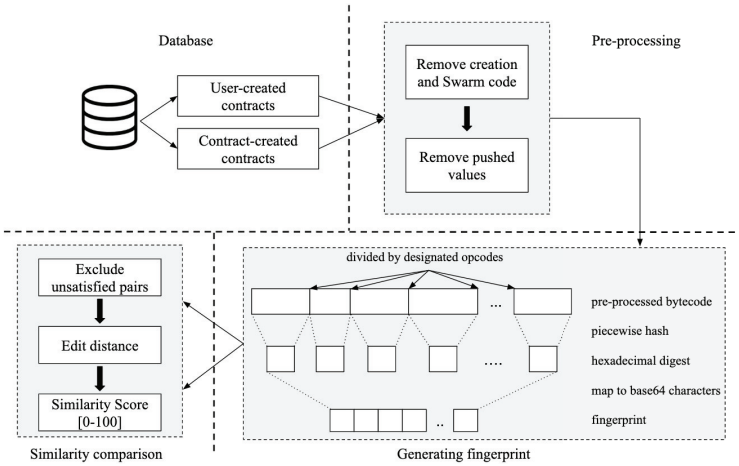


Fig. 3. An overview of our approach on smart contract similarity comparison.

As we mentioned, creation code and Swarm code have nothing to do with similarity calculation. Fortunately, they can be easily identified and removed from the bytecode. Afterwards, we use a hash set to guarantee the uniqueness of the remaining contracts in terms of runtime code. Secondly, to enable fast and accurate fingerprint generation, we further remove all the immediate numbers after opcode `PUSH` to eliminate the interference of operands. Again, we use a hash table to guarantee the uniqueness of the remaining contracts. In this way, we obtain a minimized database with little feature lost for similarity detection.

3.2 Generating Fingerprint

Calculating the edit distance between two given sequences is a well-known way to measure their similarity. In this work, we use a *fuzzy hashing* technique [61] to condense the original bytecode to a much shorter fingerprint and then calculate the edit distance between two fingerprints. Unlike traditional hash functions, fuzzy hashing first divides the bytecode sequence into smaller pieces, then uses a piece-wise hash function to perform the calculation for each piece and finally concatenates those generated piece-wise hashes to form a fingerprint. Suppose someone modifies one particular function, all the related pieces would generate different piece-wise hashes with the original ones, but the other pieces were not affected at all. In short, fuzzy hashing has advantages of accurate representation and less computing-time consumption.

However, there still exists challenges to determine the boundary of each piece. Previous work chooses a boundary randomly or simply divides the sequence by a pre-defined step (*e.g.*, seven bytes) [40]. Nevertheless, a smart contract is not just a piece of plain-text. It has semantic meaning. To address the challenge, we propose a customized fuzzy hashing algorithm, which is capable of segmenting smart contracts precisely to generate feasible piece-wise hashes.

Customized Fuzzy Hashing. After investigating the bytecode and its execution procedure in EVM, we identify the runtime code that can be further divided

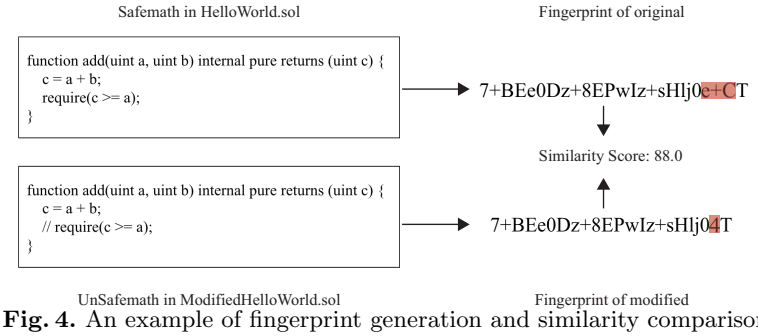


Fig. 4. An example of fingerprint generation and similarity comparison.

into several sub-sequences to perform a basic block level analysis. In Solidity, opcodes `JUMP`, `JUMPI`, `REVERT`, `STOP`, `RETURN` are the indicators of the interruption of logical relationship, and these opcodes often mean that the current block should be terminated in building the control flow graph (CFG). Futhermore, as we mentioned in Section 2, runtime code always keeps the order of function selector, function wrapper, *etc.*, and maintains the relative order between functions. After dividing, the piece-wise hash function will be applied on each of the blocks to generate a four byte hexadecimal digest and then mapped to a base-64 character after modulo 64. Finally, a fingerprint is generated by concatenating these characters (cf. Algorithm 1 in Appendix and Figure 4).

3.3 Similarity Comparison

At this stage, we are able to perform pair-wise comparison to characterize the similarity between contracts. Since pair-wise comparisons are computationally expensive (billions of comparisons), we propose a pruning strategy here to tackle the problem. Intuitively, similar contracts should share similar attributes with minor modifications (opcode length in particular). If two contracts are "very different" in the opcode length, we will stop comparing the fingerprints and mark them as dissimilar. In our implementation, if more than 30% attributes of two smart contracts are different, the comparison process will stop.

For each contract pair, we calculate the edit distance between the fingerprints, and then map it to a similarity score in the range of 0 to 100, as follows:

$$similarityScore = \left[1 - \frac{distance}{\max(len(fp1), len(fp2))}\right] * 100 \quad (1)$$

Figure 4 shows an example of the fingerprints we generated for HelloWorld.sol and its modified version, respectively. In the modified version, we have removed the *require* statement of the *SafeMath* library, which may lead to an overflow vulnerability. The difference between these two fingerprints is highlighted. Obviously, only a few characters within the fingerprint have changed, and the similarity score calculated by our approach is 88.0.

4 QUANTITATIVE ANALYSIS

In this section, we focus exclusively on quantitative analysis, which provides some straightforward but interesting findings we observed before we perform more detailed analysis in Section 5.

Table 1. An overview of the dataset before and after pre-processing.

Contract type	# Contracts (# Owned Accounts)	After Swarm code removing	After push arguments removing
user-created	2,121,745 (94,307)	105,258	74,647
contract-created	7,729,012 (29,708)	4,539	3,964

4.1 Dataset

We have collected by far the largest smart contract dataset at the time of writing, covering almost 10 million smart contracts deployed on the Ehtereum mainnet from July 30th, 2015 to December 31st, 2018. As shown in Table 1, only 2.1 million contracts are user-created, and the number of contract-created contracts is four times greater than user-created ones. They were owned by $124,015$ accounts, including $94,307$ for the user-created contracts and $29,708$ for the contract-created contracts.

4.2 Pre-processing

The pre-processing step is helpful in removing duplicates. It turns out that the proportion of contracts to be analyzed has been shrunk dramatically to $0.798\%(78,611/9,850,757)$ of the original dataset we collected. Especially for the contract-created contracts, only $3,964$ distinct contracts remained.

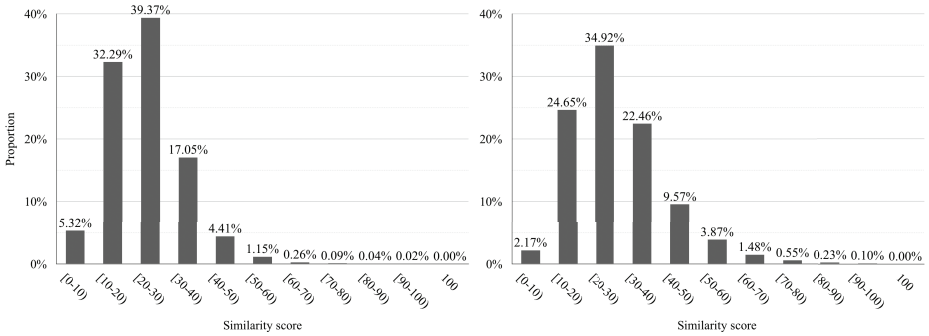
To figure out the reason for the huge number of duplicates, we first grouped the duplicated contracts into clusters, and then analyzed the distribution of those clusters (cf. Figure 8 in Appendix.) We list the top 10 contracts with the most duplicates in Table 2. It shows that the top 10 clusters represent the majority of the user-created contracts (62.37%) and the contract-created contracts (82.26%).

After further investigation, we found that most of the user-created contracts belonged to `transfer wallets`. The transfer wallet can be used in different ways, *e.g.*, avoiding regulation by initiating multiple and multilevel small transfers, which splits a large balance from one account to several seemingly irrelevant accounts. Some contracts are regarded as `forwarder`s (the contract name), which are not wallets but might be functionally similar to those transfer wallets in some way, such as transferring ETHs or tokens. Note that there are forwarders in contract-created contracts as well. Besides, some of the other duplicated contracts are controlled by exchanges, *e.g.*, Poloniex [24], to manage issued tokens, such as Golem [44] and Storj [41].

As for the contract-created contracts, some clusters are owned by the Bittrex exchange [23]. More interestingly, the second largest cluster is a token issued by Gastoken [43], which allows users to make profits by tokenizing gas based on the refund mechanism on storage in Ethereum. We also found lots of `Proxy` contracts, which were used to redirect all incoming message calls to other deployed contracts. In addition, many contracts belong to ENS [19] (Ethereum Name Service), a naming system based on the Ethereum Blockchain. Finally, there are two interesting groups related to CryptoMidwives [4], which are a kind of contracts aiming to get profit from 'CryptoServices' (*i.e.*, CryptoKitties-like games).

Table 2. Top 10 contracts with the most number of duplicates.

User-created contracts		Contract-created contracts	
# Duplicates	Use	# Duplicates	Use
390,020	Transfer wallet	1,619,511	Bittrex wallet
306,600	Transfer wallet	1,284,440	Gastoken
125,929	Transfer wallet	776,441	Bittrex wallet
123,787	Transfer wallet	544,834	Proxy
89,134	Transfer wallet	540,094	Proxy
85,782	Transfer wallet	511,894	Forwarder
68,297	Token manager of Poloniex	420,822	ENS
59,543	Token-only forwarder	277,380	CryptoMidwives
37,628	Transfer wallet	196,260	CryptoMidwives
36,625	Token manager of Poloniex	185,889	Forwarder



(a) Distribution of user-created contracts similarity score

(b) Distribution of contract-created contracts similarity score

Fig. 5. The distribution of similarity scores for smart contract pairs.

4.3 Similarity Comparison

For all the original 10 million smart contracts, it would be unfeasible for us to perform pair-wise comparison. Taking advantage of our pruning strategies, we are able to narrow down the contract pairs by almost four orders of magnitude, which greatly reduces the burden on similarity comparison.

The distribution of similarity score. With our pruning strategies, over 308 million user-created contract pairs and 1.2 million contract-created pairs were compared, and Figure 5 shows the distribution of similarity scores. Roughly 90% of the contract pairs have similarity scores less than 40, while only a small percentage of contract pairs have similarity scores higher than 70, among them are 0.153% of user-created contracts and 0.879% of contract-created contracts. In addition, 300 contract pairs have the highest similarity score, *i.e.*, 100.

Determining the threshold. First, for smart contract pairs with similarity score at different ranges (e.g., [50, 60) and [60, 70)), we randomly samples 100 pairs each range (1,100 pairs in total). Note that, to better get the ground truth, we select only the smart contract pairs with source code available. The first two authors performed manually comparison of the source code to label the ground truth. In this way, we could measure the accuracy of our approach at different thresholds. It is interesting to observe that, with a threshold of 80, our approach

could achieve a precision of 100%, while with only 66% of recall. With a threshold of 60, our approach could achieve a recall rate of 96%, while with only 86% of precision. As a result, we empirically found that 70 is a good indicator to achieve the balance, with 97.5% of precision and 88% of recall at this threshold. It is also the reason why we propose a prune strategy (cf. Section 3.3) to discard the different smart contract pairs. Note that, this threshold is inline with other fuzzy hashing based code clone detection studies [57, 74].

In the end, 472,663 user-created contracts pairs (with 47,242 contracts involved) and 11,161 contract-created contracts pairs (with 2,409 contracts involved) were considered to be similar.

5 Qualitative Analysis

Our previous observations suggest that over 96.07% of user-created contracts and 99.97% of contract-created contracts have duplicates, and a large number of contract pairs were similar. In this section, we delve deeper into qualitative evaluation. We first seek to cluster the distinct contracts into groups based on their similarity scores, for which we try to understand the reasons leading to contract clones and study the diversity of the ecosystem (*e.g.*, what are these contracts?). Then we propose to explore the correlation between code clones and vulnerabilities, *i.e.*, whether code clones lead to the spread of vulnerabilities. At last, we try to identify the DApp Clones in the wild and measure their impact.

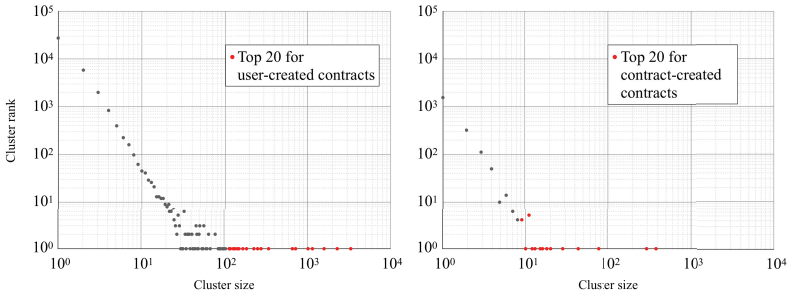
5.1 Clustering Smart Contracts

The Clustering Approach. Here, we use a simple but effective approach to cluster these contracts based on their similarity scores. Specifically, we cluster any contract pair whose similarity score is 70 or higher. Therefore, we are able to build a `weighted undirected graph` by treating each contract as a node. There will be an edge between two nodes if their similarity score (*i.e.*, weight) is larger than or equal to 70. Then, we traverse the graph and consider each `connected component` as a cluster. To sum up, only unique contracts are used to construct the graph, and only contracts with edges whose weights are higher than 70 can be regarded as a connected component to form a cluster.

Clustering Result. We apply the clustering approach on user-created contracts and contract-created contracts, respectively. The results are presented in Figure 6, which follows a long-tail distribution. For user-created contracts, over 63.29% of them were clustered into 9,971 clusters, with 27,405 isolated nodes. For contract-created contracts, 60.77% of them were clustered into 2,409 clusters.

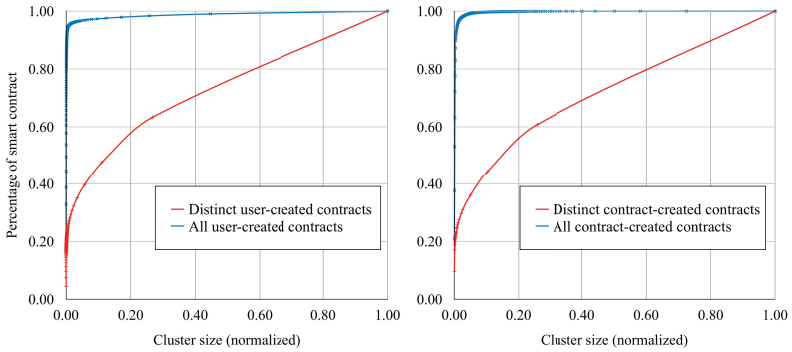
We further investigate whether these clusters follow the `Pareto principle` (*i.e.*, the 80/20 rule). The results suggest that the distribution of clusters follows a typical `Pareto Effect` after cluster size based normalization, as shown in Figure 7. For the distinct contracts, the top 20% of the clusters account for 60% of the contracts. With regards to all the contracts, including all the duplicates, the top 1% of the clusters account for over 95% of the contracts (95.24% and 95.30% for the user-created and contract-created contracts, respectively).

What are these smart contract clusters? Table 3 lists the top 10 clusters for user-created contracts and contract-created contracts. We manually went



(a) Clustering result for user-created contracts (b) Clustering result for contract-created contracts

Fig. 6. The distribution of clusters.



(a) CDF for user-created contracts (a) CDF for contract-created contracts

Fig. 7. CDF of smart contracts according to cluster sizes.

through these clusters and labelled them according to their functionalities. Each type of contracts has its own characteristic functions, *e.g.*, refund and deposit, airdrop and distribution, transfer and so on. Our exploration suggests that the largest clusters mainly fall into the following categories:

(1) ERC-20 Clusters. ERC-20 related contracts take the majority of popular clusters. We successfully identified a number of ERC-20 clusters, which might derive from different solc versions, as new versions of solc may bring in new opcodes; or more importantly, from different ERC-20 templates, as a result of different implementations of revisions of ERC-20 standard (*e.g.*, OpenZeppelin [42] libraries). By manually analyzing the top 100 clusters, we have compiled a list of 53 different templates that were widely used in smart contracts. Note that the similar contracts created by these templates were not necessarily plagiarized.

(2) Game Contracts. Many popular clusters are game contracts. The largest game cluster is `Fomo3D-like contracts`. Due to the popularity of `Fomo3D`, numerous developers just copied and pasted the original open-source contracts to create similar games. Besides Fomo3D, other popular games such as `PoWH3D` [12] and `CryptoKitties` [3], have contract clones as well.

(3) ICO and Airdrop Exploit Contracts. ICO [29] stands for Initial Coin Offering, the cryptocurrency equivalent of IPO (Initial Public Offering). It is a way for crypto startups to raise money by selling tokens. ICO has experienced an explosive growth since 2017 [32] (and the bubble burst at the end of the third

Table 3. Top 10 clusters for both user-created and contract-created contracts.

User-created contracts		Contract-created contracts	
Size (with dup)	Usage	Size (with dup)	Usage
3,338 (15,713)	ERC-20 token	382 (1,799)	ERC-20 token
2,293 (19,263)	ERC-20 token	295 (1,983)	ERC-20 token
1,596 (11,737)	ERC-20 token	76 (1,210)	ERC-20 token
1,155 (6,174)	ERC-20 token	43 (209)	ICO
1,022 (6,466)	ERC-20 token	28 (223)	ICO
724 (4,494)	ERC-20 token	20 (571)	Airdrop Exploit
662 (2,418)	ERC-20 token	18 (39)	ERC-20 token
343 (1,054)	Other contract	16 (30)	ITO
278 (972)	ERC-20 Token	15 (922)	Airdrop Exploit
253 (509)	Fomo3D-like game	13 (20)	Exchange wallet

quarter 2018), which explains why a vast number of such contracts were deployed during this period. In terms of Airdrop Exploit contracts [18], attackers have to create large numbers of these contracts to win the 'race of exploitation'.

(4) Other Contracts. We also observed that there do exist some short contracts with extremely simple operations, *e.g.*, a pair of getter and setter, fetching data from storage, *etc.*. Such contracts were grouped into clusters as well.

Observation-1: Although millions of contracts were deployed on Ethereum, most of them were duplicates and share same/similar code and functionalities, which suggested the homogeneous nature of the ecosystem.

5.2 Vulnerability Provenance

We then seek to explore the correlation between code clones and security vulnerabilities in two ways. First, we want to measure the vulnerability introduced by duplicate contracts, *i.e.*, the original contracts are suffering from vulnerabilities, and other duplicate contracts (with same hash values) would inherit the vulnerabilities. Then, for the distinct contracts that were very similar, we seek to measure whether they have the same vulnerabilities introduced by code clones.

Vulnerability Detection. To identify security vulnerabilities, we take advantage of a state-of-the-art tool [37] developed by PeckShield. It is a bytecode level static analysis framework composed of multiple program analysis techniques, including control flow analysis, data flow analysis and symbolic execution. We focus on 7 types of vulnerabilities that might cause damages with real impact, including (1) reentrancy, (2) overflow, (3) cross-function race condition, (4) mismatched constructor, (5) ownership takeover, (6) manipulable suicide address and (7) ERC-20 related vulnerabilities. As it is not the emphasis of this paper, we will use the results directly without giving technical details of the tool.

Vulnerable Duplicate Smart Contracts. We have scanned all the distinct contracts, inlcuding $74,647$ user-created and $3,964$ contract-created contracts. It is interesting to see that, although only 25K distinct user-created contracts were vulnerable, considering all the duplicate contracts, we have identified over 1.2 million vulnerable contracts. As for the contract-created contracts, the result is more striking. Only 51 unique contract-created contracts were vulnerable, but

Table 4. Distribution of vulnerability similarity across similar contract pairs.

	Same Vul behaviors		Different Vul Behaviors		
	Neither is vulnerable	Both are vulnerable	One is vulnerable	Both are vul & overlapped	Both are vul & not overlapped
Same author	26,570	3,813	6,368	1,678	247
Different author	180,101	42,368	143,146	58,338	10,034
Total	206,671	46,181	149,514	60,016	10,281

we have identified over 2.2 million vulnerable contracts when we consider all the duplicates. This result suggests that a large number of duplicate contracts would suffer from the vulnerability issues inherited from the original contracts.

Copy-paste Vulnerabilities. Then, we try to measure the copy-paste vulnerabilities from those similar contract pairs (with different hash values). For the 472K similar pairs we identified (with scores over 70), we measure the similarity in vulnerabilities between them, *i.e.*, whether they share the same types of vulnerabilities and the same number of vulnerabilities. *For contracts that share both the same types and same number of vulnerabilities, we will mark them as having exactly the same vulnerability behaviors.* Note that, we further differentiate the authors of the contracts to determine whether the contract pairs are code clones between different authors or the re-deployment from the same author. As shown in Table 4, we have classified the results into two general categories.

Same Vulnerability Behaviors. Over 53% of similar contract pairs have the same vulnerability behaviors. Over 46K contract pairs share the same vulnerabilities, and over 90% of them were created by different authors. This indicates that when someone copied the code, he/she did not know that the original contracts were vulnerable, and thus inherited the same vulnerabilities.

Different Vulnerability Behaviors. Over 46% of the similar contract pairs have different vulnerability behaviors. For over 149K contract pairs where only one contract is vulnerable, roughly 96% of them were created by different authors. It indicates that when the authors copy and paste the code, they may have identified the vulnerabilities and thus patched them. Another scenario to explain this is that their modification of the original contracts may introduce new security vulnerabilities. Besides, over 12% of the similar contract pairs were found sharing vulnerabilities, which could also be introduced by code reuse.

Case Study. Here, we use the Fomo3D-like game contracts as a case study. We have identified 253 distinct contracts belonging to this cluster. As the original Fomo3d game suffers from the "Airdrop Vulnerability [1]", over 80% (213 out of 253) of its contract clones also share the same vulnerability.

Observation-2: Copy-paste vulnerabilities were prevalent in the smart contract ecosystem, duplicate contracts and similar contract would inherit security issues from the original vulnerable ones.

5.3 Clone Detection of DApps

As Ethereum DApps are usually open-source, the plagiaristic behaviors could also be widespread. Different from the normal smart contracts, a DApp may

consist of one or more smart contracts. To measure the extent of similarity between DApps, we proposed an advanced similarity detection method.

Definition. Here, we use the term *DApp Clones* to describe the scenario where two DApps deployed by different authors share the similar core functionalities. We use the accounts to differentiate the authorship. As a large number of smart contracts were created on top of templates, thus we will first eliminate the impact introduced by the templates based on the list we labelled in Section 5.1.

Approach. For a given DApp pair, we first construct a weighted bipartite graph for them, and conduct bipartite graph matching on the graph. A bipartite graph is a graph whose vertices (contracts) can be divided into two disjoint sets U and V, such that every edge connects a vertex in U to one in V, *i.e.*, U and V are independent sets. Here, we will calculate the similarity score between contracts and take the score as the weight of the corresponding edge. Specifically, we take advantage of the *KuhnMunkres algorithm* [30] to identify the maximum matching - a set of the most edges with the following two properties: 1) no two edges share an endpoint; 2) the weight of edges must be guaranteed to be the highest. Therefore, we are able to calculate the similarity between DApps with more than one contract. As the calculation is not commutative, *i.e.*, $Sim(DApp1, DApp2) \neq Sim(DApp2, DApp1)$, we keep the higher one as the final score.

Result. We have made our best efforts to collect 2,533 DApps from well-known DApp browsers [15–17]. We also crawled related metadata, *e.g.*, category, volume, and the deployed time. Based on the definition of DApp clones, we have successfully identified 127 DApp clone pairs with 114 distinct DApps in total. We further grouped them into 41 clusters by leveraging the approach mentioned in Section 5.1. The results are shown in Figure 9 (cf. Appendices).

Impact. To measure the impact of DApp Clones, we decided to take the historical volume as the indicator to identify the potential financial losses. Even worse, the high volume often means an active market which might attract more capital inflows [45], thus it would cause more damage to the original authors.

In particular, we first analyzed all these 41 clusters and treated the earliest deployed DApp as the original one. Thus, we have 41 original DApps, and 73 plagiarized DApp clones in our dataset. Then we calculated the differences between the original volume and the plagiarized volumes.

The overall volume of the 41 original DApps is 304,797.344 ETH, while the volume of the 73 DApp clones reaches 89,565.321 ETH (more than USD 19 million on Sep 21, 2019, around 30% of the original market. The figures are diverse across the clusters by examining those clusters individually. For 18 out of the 41 clusters, the volumes of the clones are higher than those of the corresponding original DApps, with some clones attracting two to three times more volumes than the original. In Table 5 (cf. Appendices), we summarized the statistics for the top 10 clusters in Figure 9.

Observation-3: DApp clones caused great financial losses to the original DApps, which exposed a contradiction between copyright protection and the open-source nature of the Ethereum ecosystem.

6 RELATED WORK

Characterizing the Ethereum Ecosystem Several work have already been published to measure the Ethereum ecosystem [51, 67, 68]. For example, Chen *et al.* characterized money transfer, contract creation and contract invocation of Ethereum based on graph analysis [51]. Some researchers focused on financial activities on Ethereum, including the Ponzi scheme [52] and ICO behavior [56] These studies may have a correlation with part of our work, however, our work is the first systematic attempt to study contract clone phenomenon and its impact.

Program Analysis of the Smart contracts Based on program analysis techniques (*e.g.*, symbolic execution and formal verification), several frameworks have been proposed to detect vulnerabilities in contracts [33, 58, 66, 71]. Some other studies were focused on topics including reverse engineering [75], detecting gas-costly patterns [50], automatically creating exploits [62], etc. However, none of them performed a comprehensive study on the vulnerability provenance.

Code Clone Detection Code clone detection techniques have been studied extensively for dozens of years, including text-based techniques [63, 69], token-based techniques [47, 48, 59, 64], counting-based techniques [73], and syntactic approaches [49, 53, 70], etc. These techniques were also widely explored in related domains, such as mobile app repackaging detection [54, 55, 72, 74]. In this work, we take advantage of a customized fuzzy hashing technique [61], which is both light-weight and effective. A limited number of studies have explored code cloning in smart contracts. For example, Kiffer *et al.* identified substantial code reuse in Ethereum [60]. Furthermore, Liu *et al.* proposed ECLONE [65], which is able to detect semantic clones for smart contracts. However, none of them have measured the ecosystem in large-scale, and characterized their security impacts.

7 Concluding Remarks and Future Work

We present the first systematic attempt to characterize the code clone phenomenon in the Ethereum ecosystem. By analyzing the 10 million contracts, we have revealed the homogeneity nature of the ecosystem. We discovered and measured the security impacts of contract clones, e.g., helping spread the security vulnerabilities and causing financial losses to the original DApps authors, etc.

There are a number of future lines of work we will explore. First, the threshold used to identify contract clones can be improved by adopting adaptive approaches. Second, we may have coverage issues on manually labelling the contract templates, which can be alleviated by exploring some advanced techniques. Lastly, part of our findings, such as those economic intensive phenomena in the Ethereum ecosystem, deserve more focused studies. Nonetheless, we believe our efforts and observations could positively contribute to the community and promote the best operational practices for smart contract developers.

Acknowledgment

This work is supported by the National Key Research and Development Program (2017YFB1001904) and the National Natural Science Foundation of China (61702045, 61772042). Haoyu Wang and Yao Guo are co-corresponding authors.

References

1. Airdrop Vulnerability, https://blog.peckshield.com/2018/07/24/fomo3d/
2. CoinMarketCap, a Browser for Cryptocurrency Market Cap, https://coinmarketcap.com/currencies/ethereum/
3. CryptoKitties Official Website, https://www.cryptokitties.co/
4. CryptoMidwives Introduction, https://medium.com/block-science/exploring-cryptokitties-part-2-the-cryptomidwives-a0df37eb35a6
5. DApp, Crypto Countries, https://cryptocountries.io/
6. DApp, Crypto Gaming Coin, https://cyptogamingcoin.surge.sh/
7. DApp, Crypto Miner Token, https://minertoken.cloud/
8. DApp, Crypto Tubers, https://cryptotubers.co/
9. DApp, Pepe Farm, http://www.pepefarm.club
10. DApp, Po50, http://po50.surge.sh/exchange/
11. DApp, PoHD, https://pohd.io/
12. DApp, PoWH 3D, https://powh.io/
13. DApp, PoWTF, https://powtf.com/
14. DApp, Proof Of Craig Grant Coin, http://www.pocg.site
15. DAppRadar, a DApp Browser, https://dappradar.com/
16. DAppReview, a DApp Browser, https://dapp.review/
17. DAppTotal, a DApp Browser, https://dapptotal.com/
18. Definition of Airdrop Mechanism, https://coinsutra.com/what-is-airdrop/
19. Ethereum Name Service, https://ens.domains/
20. Ethereum Virtual Machine, http://ethdocs.org/en/latest/introduction/what-is-ethereum.html\#ethereum-virtual-machine
21. Ethereum White Paper, https://github.com/ethereum/wiki/wiki/White-Paper
22. Etherscan, a Ethereum Smart Contract Browser, https://etherscan.io/
23. Exchange, Bittrex, https://international.bittrex.com/
24. Exchange, Poloniex, https://poloniex.com/
25. Fomo3D Attack Event, https://blog.peckshield.com/2018/07/24/fomo3d/
26. Honeypot Smart Contract, https://medium.com/coinmonks/dissecting-an-ethereum-honey-pot-7102d7def5e0
27. Honeypot Smart Contract, https://medium.com/coinmonks/an-analysis-of-a-couple-ethereum-honeypot-contracts-5c07c95b0a8d
28. Honeypot Smart Contract, https://medium.com/coinmonks/the-phenomena-of-smart-contract-honeypots-755c1f943f7b
29. Initial Coin Offering, https://bitcoinmagazine.com/guides/what-ico
30. Kuhn-Munkres Algorithm, https://brilliant.org/wiki/hungarian-matching/
31. Last Winner Attack Event, https://medium.com/@anchain.ai/largest-smart-contract-attacks-in-blockchain-history-exposed-part-1-93b975a374d0
32. List of ICO Resources, http://startupmanagement.org/2017/03/13/the-ultimate-list-of-ico-resources-18-websites-that-track-initial-cryptocurrency-offerings/
33. Mythril, Smart Contract Analyzer, https://github.com/ConsenSys/mythril-classic
34. Official Explanation of DApp, https://github.com/ethereum/wiki/wiki/White-Paper\#ethereum

35. Official Fomo3D Contract, https://etherscan.io/address/0xa62142888aba8370742be823c1782d17a0389da1
36. Official Fomo3D Website, https://exitscam.me/
37. PeckShield, Inc. Scanner, https://peckshield.com/securityrating/scan.html
38. Smart Contract Function Signature, https://solidity.readthedocs.io/en/v0.5.3/abi-spec.html\#function-selector
39. Solidity, https://solidity.readthedocs.io/en/v0.6.0/
40. Spamsum Algorithm, https://www.samba.org/ftp/unpacked/junkcode/spamsum/README
41. Storj Token, https://storj.io/
42. Template Library, OpenZeppelin, https://github.com/OpenZeppelin/openzeppelin-solidity
43. Token, GasToken, https://gastoken.io/
44. Token, Golem, https://golem.network/
45. Trading Volume, https://m.rediff.com/money/special/trading-volume-what-it-reveals-about-the-market/20090703.htm
46. Wikipedia of Ethereum, https://en.wikipedia.org/wiki/Ethereum
47. Baker, B.S.: On finding duplication and near-duplication in large software systems. In: Proceedings of 2nd Working Conference on Reverse Engineering. pp. 86–95. IEEE (1995)
48. Baker, B.S.: Parameterized pattern matching: Algorithms and applications. Journal of computer and system sciences **52**(1), 28–42 (1996)
49. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). pp. 368–377. IEEE (1998)
50. Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 442–446. IEEE (2017)
51. Chen, T., Zhu, Y., Li, Z., Chen, J., Li, X., Luo, X., Lin, X., Zhange, X.: Understanding ethereum via graph analysis. In: IEEE INFOCOM 2018-IEEE Conference on Computer Communications. pp. 1484–1492. IEEE (2018)
52. Chen, W., Zheng, Z., Cui, J., Ngai, E., Zheng, P., Zhou, Y.: Detecting ponzi schemes on ethereum: Towards healthier blockchain technology. In: Proceedings of the 2018 World Wide Web Conference on World Wide Web. pp. 1409–1418. International World Wide Web Conferences Steering Committee (2018)
53. Corazza, A., Di Martino, S., Maggio, V., Scanniello, G.: A tree kernel based approach for clone detection. In: 2010 IEEE International Conference on Software Maintenance. pp. 1–5. IEEE (2010)
54. Crussell, J., Gibler, C., Chen, H.: Attack of the clones: Detecting cloned applications on android markets. In: European Symposium on Research in Computer Security (ESORICS). pp. 37–54. Springer (2012)
55. Crussell, J., Gibler, C., Chen, H.: Scalable semantics-based detection of similar android applications. In: European Symposium on Research in Computer Security (ESORICS). vol. 13 (2013)
56. Fenu, G., Marchesi, L., Marchesi, M., Tonelli, R.: The ico phenomenon and its relationships with ethereum smart contract environment. In: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE). pp. 26–32. IEEE (2018)
57. Glanz, L., Amann, S., Eichberg, M., Reif, M., Hermann, B., Lerch, J., Mezini, M.: Codematch: obfuscation won't conceal your repackaged app. In: Proceedings

of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 638–648. ACM (2017)
58. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: Analyzing safety of smart contracts. In: 25th Annual Network and Distributed System Security Symposium, NDSS. pp. 18–21 (2018)
59. Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering **28**(7), 654–670 (2002)
60. Kiffer, L., Levin, D., Mislove, A.: Analyzing ethereum's contract topology. In: Proceedings of the Internet Measurement Conference 2018. pp. 494–499. ACM (2018)
61. Kornblum, J.: Identifying almost identical files using context triggered piecewise hashing. Digital investigation **3**, 91–97 (2006)
62. Krupp, J., Rossow, C.: teether: Gnawing at ethereum to automatically exploit smart contracts. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 1317–1333 (2018)
63. Lee, S., Jeong, I.: Sdd: high performance code clone detection system for large scale source code. In: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. pp. 140–141. ACM (2005)
64. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: Cp-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Transactions on software Engineering **32**(3), 176–192 (2006)
65. Liu, H., Yang, Z., Liu, C., Jiang, Y., Zhao, W., Sun, J.: 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. pp. 900–903. ACM (2018)
66. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. pp. 254–269. ACM (2016)
67. Norvill, R., Pontiveros, B.B.F., State, R., Awan, I., Cullen, A.: Automated labeling of unknown contracts in ethereum. In: 2017 26th International Conference on Computer Communication and Networks (ICCCN). pp. 1–6. IEEE (2017)
68. Payette, J., Schwager, S., Murphy, J.: Characterizing the ethereum address space (2017)
69. Roy, C.K., Cordy, J.R.: Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: 2008 16th iEEE international conference on program comprehension. pp. 172–181. IEEE (2008)
70. Selim, G.M., Foo, K.C., Zou, Y.: Enhancing source-based clone detection using intermediate representation. In: 2010 17th Working Conference on Reverse Engineering. pp. 227–236. IEEE (2010)
71. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M.: Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 67–82. ACM (2018)
72. Wang, H., Guo, Y., Ma, Z., Chen, X.: Wukong: A scalable and accurate two-phase approach to android app clone detection. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. pp. 71–82. ACM (2015)
73. Yuan, Y., Guo, Y.: Boreas: an accurate and scalable token-based approach to code clone detection. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. pp. 286–289. ACM (2012)

74. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the second ACM conference on Data and Application Security and Privacy. pp. 317–326. ACM (2012)
75. Zhou, Y., Kumar, D., Bakshi, S., Mason, J., Miller, A., Bailey, M.: Erays: reverse engineering ethereum's opaque smart contracts. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 1371–1385 (2018)

8 Appendices

8.1 Fingerprint Generation Algorithm

The detailed fingerprint generation algorithm is shown in Algorithm 1.

Algorithm 1 Generating the fingerprint for smart contract.

Input: *bytecode* of arbitrary contract
Output: Fingerprint *fp*
Description: *pc* - character representing current piece, *ph* - the piece hash, *tv* - trigger value, *b64map* - mapping integer to base64 character

```
1: procedure GenerateFp(bytecode)
2:     InitTriggerValue(tv)
3:     InitBase64Map(b64map)
4:     InitPieceCharacter(pc)
5:     InitPieceHash(ph)
6:     pieces ← CutOff(bytecode, tv)
7:     for all piece from pieces do
8:         UpdatePieceHash(ph, piece)
9:         MapToPieceCharacter(pc, ph, b64map)
10:        fp ← Concatenate(fp, pc)
11:        InitPieceHash(ph)
12:    return fp
```

8.2 The distribution of contract clusters grouped by opcode hash values

Figure 8 shows the distribution of contract clusters grouped by opcode hash values.

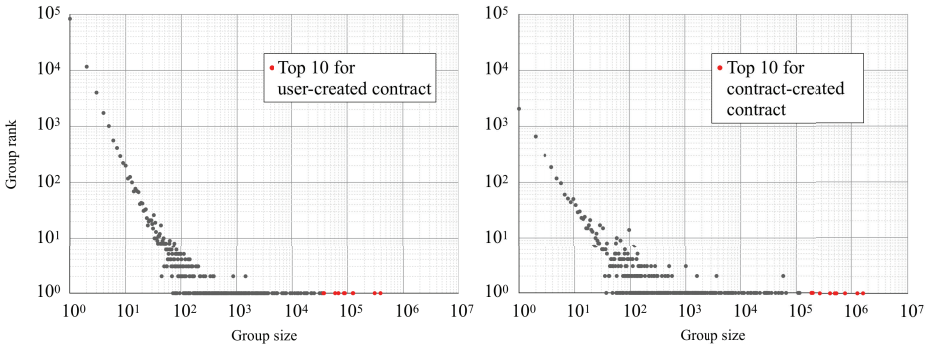


Fig. 8. The distribution of contract clusters grouped by opcode hash values.

8.3 Clustering results of the 127 Dapp Clone Pairs

Figure 9 shows the result of the clustering results of the 127 Dapp clone pairs we identified.

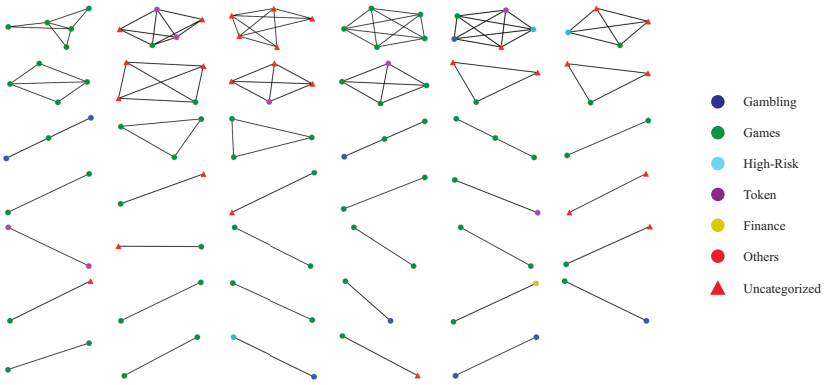


Fig. 9. Clustering results of 127 DApp clone pairs (114 unique DApps).

8.4 Top Dapp Clone Clusters and Their Volumes

Table 5 lists the statistics of the top 10 Dapp clone clusters.

Table 5. Top 10 Dapp Clone clusters and their volumes (ETH).

Original DApp	# Clones	Original volume	Plagiarized volume	Ratio
CryptoCountries [5]	4	67,885.244	2.355	<0.01%
PoWTF [13]	4	331.074	1,012.649	305.87%
Po50 [10]	4	76.801	213.058	277.42%
Pepe Farm [9]	4	25.428	33.577	132.05%
Crypto Miner [7]	4	17,312.026	155.437	0.90%
PoWH 3D [12]	4	187,950.872	1,778.146	0.95%
CryptoTubers [8]	3	95.378	470.967	493.79%
PoHD [11]	3	242.607	5,867.961	2418.71%
Proof Of Craig Grant Coin [14]	3	642.056	94.315	14.69%
Crypto Gaming Coin [6]	3	4.711	555.142	11783.95%