ECLONE: Detect Semantic Clones in Ethereum via Symbolic Transaction Sketch

Han Liu [*†]
School of Software
Tsinghua University
Beijing, China
liuhan2017@tsinghua.edu.cn

Zhiqiang Yang
School of Software
Tsinghua University
Beijing, China
softwareyangzq12@163.com

Chao Liu
Peking University
Beijing, China
liuchao_cs@pku.edu.cn

Yu Jiang[‡]
School of Software
Tsinghua University
Beijing, China
jy1989@tsinghua.edu.cn

Wenqi Zhao
Ant Fortune Business Group
Ant Financial
Beijing, China
muhan.zwq@antfin.com

Jiaguang Sun
School of Software
Tsinghua University
Beijing, China
sunjg@tsinghua.edu.cn

ABSTRACT

The Ethereum ecosystem has created a prosperity of smart contract applications in public blockchains, with transparent, traceable and programmable transactions. However, the flexibility that everybody can write and deploy smart contracts on Ethereum causes a large collection of similar contracts, *i.e.*, *clones*. In practice, smart contract clones may amplify severe threats like security attacks, resource waste *etc.*.

In this paper, we have developed ECLONE, a semantic clone detector for Ethereum. The key insight of our clone detection is *Symbolic Transaction Sketch*, *i.e.*, a set of critical semantic properties generated from symbolic transaction. Sketches of two smart contracts will be normalized into numeric vectors with a same length. Then, the clone detection problem is modeled as a similarity computation process where sketches and other syntactic information are combined. We have applied ECLONE in identifying semantic clones of deployed Ethereum smart contracts and achieved an accuracy of 93.27%.

A demo video of ECLONE is at https://youtu.be/IRasOVv6vyc.

CCS CONCEPTS

• **Software and its engineering** → **Reusability**; *Software verification and validation*;

KEYWORDS

Smart contracts, semantic clone, symbolic transaction

ACM Reference Format:
Han Liu, Zhiqiang Yang, Chao Liu, Yu Jiang, Wenqi Zhao, and Jiaguang Sun. 2018. ECLONE: Detect Semantic Clones in Ethereum via Symbolic Transaction Sketch. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3236024.3264596

1 INTRODUCTION

Smart contracts, a special form of programs on blockchain, were first introduced by Ethereum [16] in order to enable transparent, traceable and programmable transactions. Using high-level programming languages such as Solidity [5], developers can implement complex business logic in smart contracts. Figure 1 shows a simple Solidity smart contract, which defines a cryptocurrency token called `FSECoin`. As traditional programs, this contract declares a variable `balances` whose scope covers the whole contract. Specially, `balances` is called *state variables* and permanently stored on blockchain. Furthermore, a `transfer` function is defined to transfer tokens between two addresses. Particularly, smart contracts will be compiled into Ethereum Virtual Machine (EVM) bytecode [16] and assigned with specific blockchain addresses. Other Ethereum accounts can call a smart contract by sending transactions to its address, specifying which function is called and what argument values are passed.

```
1  contract FSECoin ...
2    mapping (address=>uint) public balances;
3    function transfer (address recv, uint amount) {
4      if(balances[msg.sender] < amount) throw;
5      balances[msg.sender] -= amount;
6      balances[recv] += amount;
```

Figure 1: A simple Solidity smart contract

[*] Also with Beijing National Research Center for Information Science and Technology.
[†] Also with Key Laboratory for Information System Security, Ministry of Education.
[‡] Correspondence author

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3264596

```
dup1    add
sload   swap1
swap1   sstore
swap2
```

(a) `--optimize`

```
dup3    swap3   swap1
dup3    pop     sstore
sload   pop     pop
add     dup2
```

(b) No `--optimize`

Figure 2: Smart contract clones of Figure 1

As other types of programs, smart contracts tend to follow the programming naturalness [8], which may lead to many similar contract code, *i.e.*, *clones*. Figure 2 displays a pair of clones (disassembled into opcodes) of line 6 in Figure 1, which are compiled with and without optimization option of the `0.4.18 solc` compiler [5] respectively. While both code perform a state variable addition, the optimized code is shorter by removing several instructions (*e.g.*, `DUP`, `POP`). In the context of blockchain, while clone detection of smart contracts can enable important applications such as vulnerability discovery (find clones of known vulnerable contracts) and deployment optimization (reduce contract size), it is relatively little discussed. The main challenges of detecting semantic smart contract clones in Ethereum are summarized below.

Challenge 1: Handle Syntactic Diversity. Ethereum uses "gas" (*i.e.*, a form of fee) to charge the deployment and execution of smart contracts. Consequently, smart contracts become more syntactically diverse due to the tradeoff between reducing deployment gas and runtime gas respectively.

Challenge 2: Understand High-level Semantics. Detecting clones requires understanding the programming intents of smart contracts, *e.g.*, what kinds of transaction a contract is programmed to process. However, such high-level semantics is hard to analyze.

Our Insight. To address these challenges, we have proposed *symbolic transaction sketch* (STS) in this paper to effectively encapsulate high-level semantics of a smart contract and enable efficient clone detection as well. Generally, STS is a set of critical semantic properties (*e.g.*, path condition of money transfer, access patterns of permanent storage *etc.*) generated from symbolic transaction, *i.e.*, executing a smart contract with symbolic transaction input. STS is then normalized into a numeric vector and combined with other syntactic information for similarity computation. Similar contracts will be labeled as semantic clones. We have implemented STS for Ethereum, which showed its potential in effectively finding real clones.

2 SYMBOLIC TRANSACTION SKETCH

Figure 3 shows the general work flow of the proposed clone detection process based on *symbolic transaction sketch* (STS). Given the EVM bytecode of two Ethereum smart contracts A and B, STS first performs semantic generation to model semantics of A and B as numeric sketch and feature vectors. Specifically, the generation involves two steps, *i.e.*, symbolic transaction where contracts are symbolically executed with symbolic values and syntactic feature extraction where specific syntactic information (*e.g.*, type and number of instructions) is extracted. With sketch and feature vectors combined, STS enforces a similarity computation process on A and B and identifies clones (§2.2).

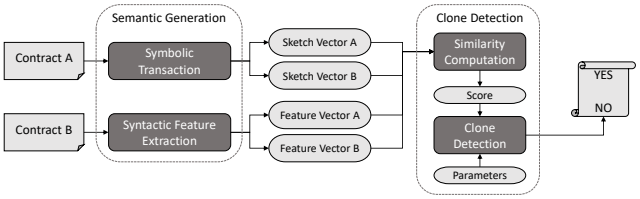


Figure 3: The semantic clone detection framework.

2.1 Semantic Generation

Assuming $\sigma = s_1 s_2 \cdots s_n$ is a runtime trace of a transaction, where s_i $(1 \leq i \leq n)$ is a specific instruction, STS is a set $\mathcal{P}$ of critical semantic properties generated from σ. Particularly, we consider a smart contract as a control flow graph (CFG) and each node in CFG is a basic block consisting of a sequence of instructions. The semantics properties are collected at basic block level. Let b be a basic block, we consider three types of properties, *i.e.*, path condition to reach b, storage accesses in b and message calls in b.

To generate the aforementioned properties from a smart contract c, we perform *symbolic transaction* on c, which is realized via symbolic execution with symbolic values, *i.e.*, nonce, gas price, gas limit, receiver address, money value attached, signature of sender and call data [16]. The call data specifies a function to call and the argument values passed to that function. At runtime, EVM instructions in c are interpreted as common symbolic execution engines like CUTE [15] and Klee [1]. When stepping into a basic block b, we compute the path conditions (PC) of b by collecting constraints on `JUMP` and `JUMPI` instructions in the path before b. As for the execution of b, we focus on three types of instructions, *i.e.*, `SLOAD` (load state variable values), `SSTORE` (store values to state variables) and `CALL` (send messages to an address). Specifically, we consider eight semantic properties (given v is a state variable): L, S and C for single `SLOAD`, `SSTORE` and `CALL`; DU for `SSTORE v; SLOAD v`; UU for `SLOAD v; SSTORE v`; UpC for `SSTORE v; CALL`; UsC for `SLOAD v; CALL`; CF for `CALL; SSTORE v`.

Formally, the critical semantic properties of an STS is defined as $\mathcal{P} = \{PC, L, S, C, DU, UU, UpC, UsC, CF\}$. Furthermore, we generate a numeric vector $\bar{\mathcal{P}}$ for $\mathcal{P}$. For PC , L, S and C in $\mathcal{P}$, we count the number of constraints and operations during symbolic transaction. In terms of the patterns, we calculate the maximal number of patterns on a single state variable and record the total number of patterns as well. Then we use cantor pairing function[1] to encode these two values (count and maximum) into a single value. On the other hand, we extract syntactic information from b as well. Specifically, we categorize instructions into six classes based on [16], *i.e.*, arithmetic, logic, environment, blockchain, stack and memory operations. Similarly, we generate a numeric vector $\bar{\mathcal{F}}$ for the basic block b by counting the number of operations in each class.

[1] https://en.wikipedia.org/wiki/Pairing_function

2.2 Clone Detection

Based on the generated semantic vectors, *i.e.*, $\bar{\mathcal{P}}$ and $\bar{\mathcal{F}}$, clone detection of Ethereum smart contracts is modeled as similarity computation problem over the vectors. Given two vectors P and Q, we define their distance $\|PQ\|$ as in Equation Vector Distance.

$$\|PQ\| = \frac{\sum \alpha_i \, |P_i - Q_i|}{\sum \alpha_i \max(P_i, Q_i)} \qquad \text{(Vector Distance)}$$

We adopt the definition as in [4]. Specifically, larger distance indicates lower similarity between two vectors. Distances between sketch ($\|\|_S$) and feature ($\|\|_F$) vectors can be calculated using this formula. Furthermore, for two basic blocks b_1 and b_2, their similarity $\|b_1 b_2\|$ is calculated in Block Similarity.

$$\|b_1 b_2\| = \frac{1 - \|\bar{\mathcal{P}}_1 \bar{\mathcal{P}}_2\|_S}{1 + \|\bar{\mathcal{P}}_1 \bar{\mathcal{P}}_2\|_S + \omega \cdot \|\bar{\mathcal{F}}_1 \bar{\mathcal{F}}_2\|_F} \qquad \text{(Block Similarity)}$$

Based on the block similarity, we denote $P(b_1; b_2)$ to be the probability measurement that b_1 and b_2 are semantic clones, *i.e.*, b_1 is semantically equivalent or similar to b_2. The probability is computed as in Clone Blocks.

$$P(b_1; b_2) = 1/(1 + e^{-k \cdot (\|b_1 b_2\| - 0.5)}) \qquad \text{(Clone Blocks)}$$

The probability is estimated by applying a sigmoid function with a midpoint to be 0.5 as $\|b_1 b_2\| \in [0, 1]$ [3]. Given two CFGs G_1 and G_2 from a pair of smart contracts, the similarity computation is realized by finding the best match in G_2 for each of the basic block of G_1 and vice versa. Moreover, searching for the best match is done by finding a pair of basic blocks with the smallest sketch vector distance. When two blocks are found, we compute their clone probability (with feature vector distance involved). Lastly, we define the semantic similarity $Sim(C_1, C_2)$ of two smart contracts C_1 and C_2 via the Clone Contracts function.

$$Sim(C_1, C_2) = \sum_{b_i \in C_1} \log \frac{P(b_i; b^*)}{P(b_i; H_0)} \qquad \text{(Clone Contracts)}$$

In particular, given a specific basic block $b_i \in C_1$, $b^* \in C_2$ and $b^* = \text{argmax} \|b_i b_j\|_S$. $P(b_i; H_0)$ represents a probability estimation of clones between b_i and a random basic block H_0. Lastly, by specifying a threshold value of $(Sim(C_1, C_1) - Sim(C_1, C_2))/Sim(C_1, C_1)$, we can filter clone contracts out of unrelated ones.

3 USING ECLONE

3.1 Architecture Design

We have implemented a clone detector called ECLONE, whose architecture is shown in Figure 4. Specifically, ECLONE provides a web service to interact with users as in Figure 5. Currently, both Solidity source code and EVM bytecode contracts are allowed. In the backend, ECLONE uses the `solc` compiler [5] to generate EVM bytecode. The *CFG Builder* then creates a control flow graph (CFG). Based on the CFG, *Feature Extractor* produces syntactic features automatically. ECLONE reuses the *Symbolic Execution Engine* from Oyente [13] to perform symbolic transaction and record semantic metadata. Then, *Semantic Vector Generator* is responsible for generating semantic vectors, which are used by *Analysis Core* to compute similarity and further check for clones.

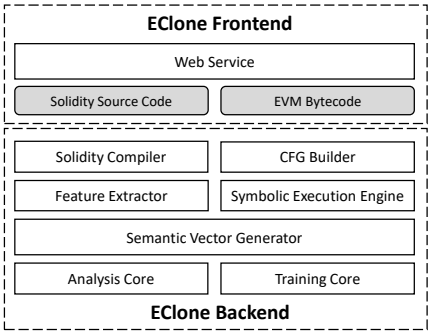


Figure 4: The architecture of ECLONE.

Additionally, ECLONE also has a *Training Core*, which is leveraged to train and optimize the parameters used in the clone detection, *e.g.*, α, ω, k as mentioned above. To this end, we prepare a corpus C of training inputs for ECLONE. Each input contains a pair of EVM bytecode with a label from $\{-1, 1\}$, where -1 means unrelated contracts and 1 means clones. Then, ECLONE employs `pyGAlib`[2] to optimize the Objective Function.

$$\max \sum (Sim(C_i, C_j) - Sim(C_i, U_k)) \qquad \text{(Objective Function)}$$

Specifically, C_i and C_j are labeled as 1 and C_i and U_k are labeled as -1 in the training data ($C_i, C_j, U_k \in C$). Conceptually, the objective function helps ECLONE separate clone and unrelated contracts as much as possible.

3.2 Key Capabilities

Figure 5 shows a screenshot of ECLONE. The web service allows users to input a pair of smart contracts, *i.e.*, Query and Target. In this case, we use two compiled versions of the `BEC` token contracts as query and target respectively, as in Figure 6. Specifically, a multiplication overflow can cause incorrect token transfer. The clone detection is enabled by clicking the analysis button at bottom.

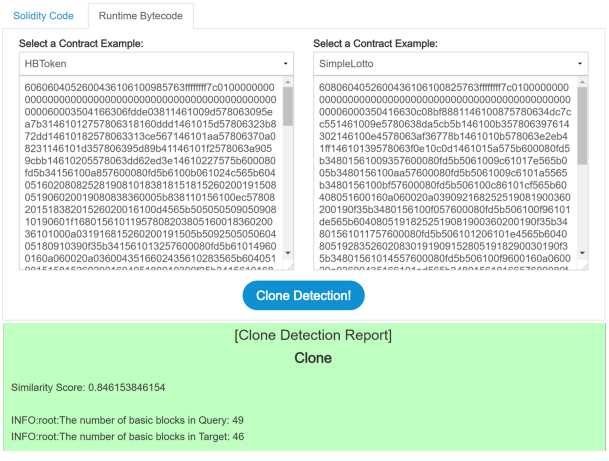


Figure 5: The architecture of ECLONE.

[2] https://github.com/gorkazl/pyGAlib

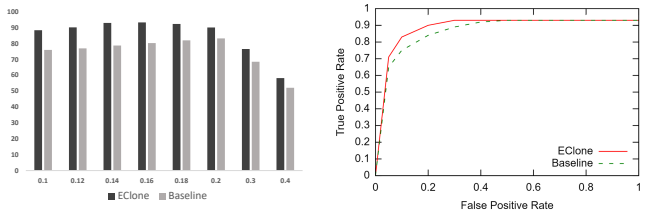
```
1    uint256 amount = uint256(cnt) * _value;
2    require(cnt > 0 && cnt <= 20);
```

Figure 6: A vulnerable fragment of BEC

Figure 5 shows the clone detection report of EClone for the contracts given in Figure 5. Specifically, EClone identifies the two smart contracts given as clones. In the Figure 5 case, the target and query are from the same BEC contract thus are semantically equivalent. Moreover, the report explains the detection details, *i.e.*, the similarity score value generated by EClone for the given smart contracts and statistics of both contracts. In this example, the query bytecode has 49 basic blocks in CFG while the target bytecode has 46. In the future, we plan to refine the report to help users better understand the contract in comparison.

4 PRELIMINARY EVALUATION

The evaluation dataset includes 2,117 Solidity smart contracts. For each contract, we use the `0.4.18 solc` compiler to compile the source file with and without `--optimize` option. From the dataset, we randomly pick 1,517 pairs for evaluation. Particularly, if two contacts of a pair are from the same source file, it was marked as clone. Otherwise, they were marked as unrelated. We compared EClone with a baseline technique, *i.e.*, using only numeric syntactic features to detect clones. Figure 7 shows the clone detection results in the evaluation.



(a) Precision of clone detection **(b) ROC curve of clone detection**

Figure 7: Clone detection results

Specifically, we mainly investigate the detection accuracy in Figure 7a. Correct labeling on both the clone and unrelated groups are considered as accurate detection. Others were counted as false labels. For different values of threshold, EClone managed to achieve a best accuracy of 93.27%, which is 12.09% better than the baseline. Moreover, we compared the ROC curve of the two techniques, as in Figure 7b. From the curves, we can tell that EClone reached a better balance between true and false positives than the baseline.

5 RELATED WORK

Clone detection has been a long-standing research problem at both source-code level [7, 9, 10] and binary level [3, 4, 6, 14, 17]. Jiang *et al.* proposed Deckard and highlighted an effective tree similarity based technique to search for matched clone pairs [9]. Gabel *et al.* further extended Deckard through mapping selected program dependence graphs [7]. In the context of binary code, Saebjornsen *et al.* extended the tree-based technique by normalizing assembly instructions with structure information considered [14]. David *et al.* further involved input-output equivalence to check semantic similarity [3]. On the other hand, machine learning techniques are applied to infer a semantic embeddings of code and further find clones [2, 17]. Smart contract analysis has been attracting research interests across various topics [11–13]. We made the first step in this area by proposing a semantic-aware clone detection technique for Ethereum.

6 CONCLUSION

In this paper, we have presented the *symbolic transaction sketch* technique and EClone tool for clone detection in Ethereum. The key insight behind EClone is to sketch out semantics of smart contracts as numeric vectors via symbolically executing a transaction. The preliminary evaluation showed the potential of EClone in accurately finding smart contract clones. A demo video is available at https://youtu.be/IRasOVv6vyc.

ACKNOWLEDGMENT

This work is sponsored by NSFC under Grant No.: 61527812, MIIT IT funds (Research and application of TCN key technologies) of China, National Science and Technology Major Project under Grant No.: 2016ZX01038101, National Key Technology R&D Program under Grant No.: 2015BAG14B01-02, and China Postdoctoral Science Foundation under Grant No.: 2017M620785.

REFERENCES

[1] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.
[2] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative embeddings of latent variable models for structured data. In *ICML*. 2702–2711.
[3] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. *ACM SIGPLAN Notices* 51, 6 (2016), 266–280.
[4] Sebastian Eschweiler and et al. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code.. In *NDSS*.
[5] Ethereum. 2017. Solidity — Solidity 0.4.19 documentation. https://solidity.readthedocs.io/en/develop/
[6] Qian Feng and et al. 2016. Scalable graph-based bug search for firmware images. In *CCS*. ACM, 480–491.
[7] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*. ACM, 321–330.
[8] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *ICSE*. IEEE, 837–847.
[9] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE*. IEEE Computer Society, 96–105.
[10] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code.. In *OSdi*, Vol. 4. 289–302.
[11] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. ReGuard: finding reentrancy bugs in smart contracts. In *ICSE (Companion)*. ACM, 65–68.
[12] Han Liu, Chao Liu, Wenqi Zhao, Yu Jiang, and Jiaguang Sun. 2018. S-gram: towards semantic-aware security auditing for Ethereum smart contracts. In *ASE*. ACM, 814–819.
[13] Loi Luu and et al. 2016. Making smart contracts smarter. In *CCS*. ACM, 254–269.
[14] Andreas Sæbjørnsen and et al. 2009. Detecting code clones in binary executables. In *ISSTA*. ACM, 117–128.
[15] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *ESEC/FSE*, Vol. 30. ACM, 263–272.
[16] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014).
[17] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *CCS*. ACM, 363–376.