Enabling Clone Detection For Ethereum via Smart Contract Birthmarks

***Abstract*—The Ethereum ecosystem has introduced a pervasive blockchain platform with programmable transactions. Everyone is allowed to develop and deploy smart contracts. Such flexibility can lead to a large collection of similar contracts, *i.e.*, *clones*, especially when Ethereum applications are highly domain-specific and may share similar functionalities within the same domain, *e.g.*, token contracts often provide interfaces for money transfer and balance inquiry. While smart contract clones have a wide range of impact across different applications, *e.g.*, security, they are relatively little studied.**

Although clone detection has been a long-standing research topic, blockchain smart contracts introduce new challenges, *e.g.*, syntactic diversity due to trade-off between storage and execution, understanding high-level business logic *etc.*. In this paper, we highlighted the very first attempt to clone detection of Ethereum smart contracts. To overcome the new challenges, we introduce the concept of smart contract *birthmark*, *i.e.*, a semantic-preserving and computable representation for smart contract bytecode. The birthmark captures high-level semantics by effectively sketching symbolic execution traces (*e.g.*, data access dependencies, path conditions) and maintain syntactic regularities (*e.g.*, type and number of instructions) as well. Then, the clone detection problem is reduced to a computation of statistical similarity between two contract birthmarks. We have implemented a clone detector called *EClone* and evaluated it on Ethereum. The empirical results demonstrated the potential of *EClone* in accurately identifying clones. We have also extended *EClone* for vulnerability search and managed to detect `CVE-2018-10376` instances.

***Index Terms*—Ethereum, clone detection, smart contract birthmark, symbolic execution**

I. Introduction

As a special form of programs on blockchain, smart contracts has been witnessing its prosperity since it was first introduced by Ethereum [1]. Smart contracts run exactly as programmed to enable transparent and traceable transactions. In Ethereum, developers are allowed to develop their own smart contracts using high-level programming languages such as Solidity [2], then deploy the contracts on Ethereum for specific business services, *e.g.*, banking services, insurance, property management, gaming *etc.*. Figure 1 shows a simple Solidity smart contract, which defines a cryptocurrency token called `Token`. As traditional programs, this contract declares a mapping-type variable `balances` whose scope covers the whole contract. Unlike memory-stored variables, smart contract variables are called *state variables* and permanently stored on blockchain. That said, any modification on `balances` will be seen in following executions. Furthermore, a `transfer` function is defined with two arguments to transfer a specific amount of cryptocurrency tokens from one account address to the other. Instead of storing source code of smart contracts on Ethereum, developers compile smart contracts, *e.g.*, `Token`, into Ethereum Virtual Machine (EVM) bytecode [1] and further deploy the bytecode onto Ethereum. Particularly, every smart contract application is assigned a 20-byte Ethereum address. Other Ethereum accounts can call a smart contract by sending a transaction to its address, specifying which function is called and what argument values are passed.

```
1  contract Token {
2    mapping (address=>uint) public balances;
3
4    function transfer (address recv, uint amount) {
5      if(balances[msg.sender] < amount)
6        throw;
7      balances[msg.sender] -= amount;
8      balances[recv] += amount;
9    }
10 }
```

Fig. 1: A simple Solidity smart contract

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

(a) `--optimize`

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

(b) No `--optimize`

Fig. 2: Smart contract clones of Figure 1

In the context of Ethereum, smart contracts are highly domain-specific. That is, contracts serving the same application domain are very likely to share similar functionalities. For example, token smart contracts often provide users with `transfer` (transfer tokens between accounts) and `balance` (check the balance of an account) interfaces. In addition, since Ethereum smart contracts are manually developed (sometimes copied and pasted) at the current stage, they tend to be quite repetitive and follow the programming naturalness [3]. Consequently, both reasons may lead to many similar contract code, which we call "*clones*". We use the example in Figure 1 to explain smart contract clones. Figure 2 demonstrates a pair of clones (disassembled into opcodes) based on Figure 1, which are compiled with and without the optimization option of the `0.4.18 solc` compiler [2] respectively. Only instructions from line 8 (*i.e.*, increase the balance of the receiver address) are shown here due to the space limit. At the first glance, it is obvious that the two clones are syntactically different. Although they are known to implement the same functionality, *i.e.*, balance update, the optimized code is shorter by removing several instructions (*e.g.*, `DUP`, `POP`). While clone detection in general-purpose software has been a long-standing research topic, it is relatively little discussed in the context of blockchain. In practice, detecting smart contract clones can enable important applications such as vulnerability discovery (find clones of known vulnerable contracts) and deployment optimization (reduce contract size by removing duplicate clones). However, the uniqueness of Ethereum blockchain introduces new challenges in clone detection of smart contracts, which are summarized as below.

Challenge 1: Tame Diversity of EVM Bytecode. Although Ethereum smart contracts may share similar programming patterns at source code level, they are syntactically diverse at EVM bytecode level. The reasons are twofold. First, compilers evolves quickly at the current stage, making the bytecode different even for the same source code. Second, Ethereum uses "gas" (*i.e.*, a form of fee) to charge the deployment and execution of smart contracts. Consequently, the contract bytecode is largely dependent on whether compiler chooses to reduce deployment or execution gas. Such diversity increases the complexity of finding contract clones.

Challenge 2: Understand Business Logics. On the other hand, associating two smart contracts requires understanding on their high-level business logics, *e.g.*, an `ERC20` token contract is designed to manage authorized operations of cryptocurrencies. In practice, such intents are often specified by the people who design the contracts, *e.g.*, the corresponding company that defines and releases a token. Without any specification, it is hard to automatically infer such high-level semantics and further effectively detect clones.

Birthmark-based Clone Detection. To address these challenges, we introduced the notion of smart contract "*birthmark*" and further proposed an EVM bytecode level clone detection technique based on birthmark. Intuitively, a birthmark is an abstract representation of a smart contract and describes its important design patterns, *e.g.*, how the contract processes different transaction requests. Specifically, a birthmark is realized as a set of numeric vectors, each of which maps to a basic node in the control-flow graph (CFG) of a smart contract and is consisted of two parts of *metadata*, *i.e.*, syntactic bytecode metadata (*e.g.*, statistics on bytecode instructions) and transaction sketch metadata (*e.g.*, pre-defined semantic patterns) respectively. Given a pair of smart contract p and q, our birthmark-based clone detection employs symbolic execution to explore the program paths in them and automatically generates the birthmarks of both p and q. Then, a statistical similarity is computed via a *best-match* algorithm, *i.e.*, finding a statistical perfect match in q for every CFG node in p and vise versa. Clones are identified by checking whether the similarity value exceeds a threshold or not.

We have developed a smart contract clone detector *EClone* to implement the birthmark-based clone detection technique, and further evaluated it on Ethereum. The empirical results demonstrated the potential of *EClone* in accurately recognizing semantic clones while distinguishing irrelevant contracts, even if they incur big syntactic noise in some cases. We also conducted an application using recognized clones, *i.e.*, vulnerability search. *EClone* has shown its practical value via efficiently finding the `CVE-2018-10376` vulnerability.

Contribution. We summarize our main contributions below.

- We have introduced the concept of smart contract *birthmark*, which is an effective and computable representation to abstract EVM bytecode and model its business logics.
- We have proposed an EVM bytecode-level birthmark-based clone detection technique for Ethereum, which leverages symbolic execution to generate birthmarks and identifies clones via computing statistical similarities.
- We have conducted a large-scale evaluation on Ethereum and for the first time discussed the smart contract clones in the current ecosystem.
- We have highlighted the application of vulnerability search based on smart contract clone detection, which has not been considered before.

Paper Organization. The remainder of the paper is organized as follows. §II introduces background information of Ethereum blockchain and EVM. §III describes the birthmark-based clone detection technique in detail. §IV demonstrates the conducted industrial evaluation. §V discusses related works and §VI concludes the whole paper.

II. Background

A. Ethereum Blockchain

Ethereum can be seen as a decentralized network consisting of two types of nodes, *i.e.*, externally owned accounts (EOA) and smart contract accounts. Every account node is assigned with a 160-bit address and associated with its own state. Specifically, the state information of EOA contains a *nonce* (number of transactions on the account) and *balance* of *ether* (the cryptocurrency in Ethereum). In terms of smart contract accounts, their states also contain storage data which is persistently stored on blockchain and smart contract code, *i.e.*, Ethereum virtual machine bytecode which we later explain.

In the Ethereum network, external actors (*e.g.*, individual users or entities) are allowed to submit cryptographically-signed transactions. Specifically, there are two types of transactions in Ethereum, *i.e.*, smart contract creation which aims at putting contract code on blockchain and message call that passes data between different accounts. Both types will be charged via *gas*, *i.e.*, a form of transaction fee in Ethereum. Transactions will be collected into blocks by mutually distrusting miner nodes and further validated. Based on a consensus protocol, *i.e.*, currently proof-of-work (PoW) in Ethereum [4], miners will agree on whose block can be merged to the blockchain. More specifically, a transaction specifies a set of common fields, including *nonce* holding the total number of transactions sent by the sender, *gasPrice* which means the price per gas, *gasLimit* which denotes the maximal amount of gas allowed to process the transaction, *to* referring to the transaction recipient address, *value* that is ether transferred to the destination account, *v,r,s* relating to the signature of the transaction and sender. Moreover, the contract creation transaction is associated with *init* filed, the contract bytecode. Instead, message calls include *data*, which specifies what function in the smart contract is called and the corresponding argument values.

B. Ethereum Virtual Machine

The execution of smart contracts happens in the Ethereum virtual machine (EVM). Particularly, EVM takes bytecode as input and works in a stack-based architecture with a word size of 256 bits. There are three different space in EVM to

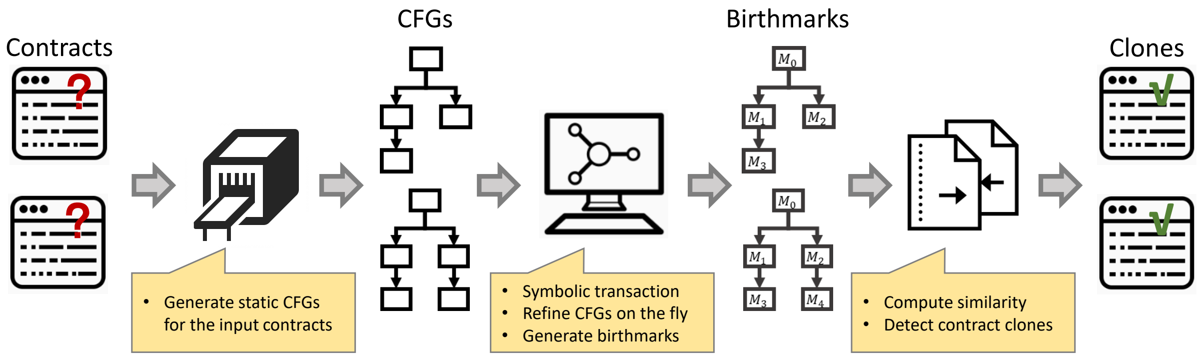


Fig. 3: The general work flow of birthmark-based clone detection ($\mathcal{BCD}$) framework.

store data and resources, namely *stack*, *memory* and *storage*. Specifically, stack holds 256-bit data which may carry different types of values. Memory is linear and can be addressed at byte level. Storage is a key-value space which maps 256-bit words to 256-bit words and maintains persistent data, *e.g.*, the `balances` state variable in Figure 1. To execute a smart contract, EVM iteratively fetches an instruction from the bytecode and operate on stack, memory and storage. According to the Ethereum yellow paper [1], there are 12 defined classes of instructions. We informally explain 3 of them, which are closely related to the technique proposed in this paper. `SSOTRE/SLOAD` stores value and loads value from the storage respectively. `CALL/CALLCODE/DELEGATECALL` are responsible for sending message calls. For `CALL` instruction, it specifies 7 parameters, *i.e.*, gas value given for the call, recipient address, ether value attached with the call, input offset, input length, output offset and output length. When executing a `CALL`, seven values are popped out from the stack for the corresponding parameters. `JUMP/JUMPI` causes a jumping operation from the current instruction to a specific offset. The destination of the jump is either unconditional (`JUMP`) or conditional (`JUMPI`). The aforementioned three types of instructions are used to capture high-level semantics of smart contracts, which will be explained later. Other instructions, *e.g.*, arithmetic operations (`ADD`, `SUB` *etc.*) and stack operations (`PUSH`, `POP` *etc.*) are considered in the proposed clone detection technique as well.

III. Birthmark-based Clone Detection

In this section, we will describe the birthmark-based clone detection ($\mathcal{BCD}$) technique for Ethereum smart contracts. Figure 3 specifies the general work flow of the $\mathcal{BCD}$ framework. Specifically, the framework takes as input a pair of smart contracts A and B, which are given in the form of EVM bytecode. The goal of $\mathcal{BCD}$ is to *quantitatively* answer the question "*Is A semantically similar to B?*". We formalize the concept of *birthmark* in §III-A. Then in order to answer this question, $\mathcal{BCD}$ first constructs a pair of static control flow graphs (CFG) based on the input bytecode. Then, it performs *symbolic transaction* (§III-B) to symbolically execute the CFG pair and refine CFGs on the fly. Intuitively, the procedure can been seen as a process of symbolic execution in the runtime of Ethereum with a set of blockchain variables. The outputs of symbolic transaction are two groups of *birthmark* vectors for A and B. Particularly, a birthmark captures not only syntactic features but semantic patterns as well, and it offers an easy-to-compute representation of Ethereum smart contracts for clone detection. Furthermore, based on the birthmarks, $\mathcal{BCD}$ runs a similarity computation algorithm to statistically compute a score that quantifies "*How similar A and B are?*". Lastly, $\mathcal{BCD}$ leverages the score and a set of configuration parameters to determine whether A and B are clones or not (§III-C).

A. Preliminaries

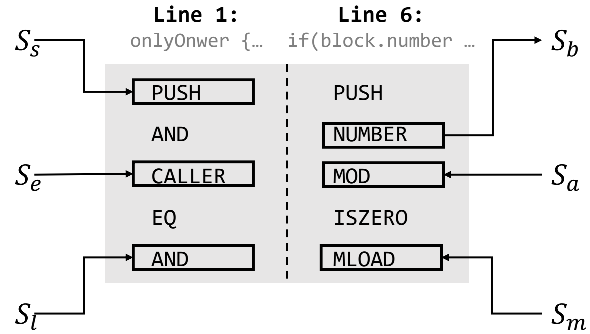
Given a smart contract bytecode s, we use the notion $\mathcal{G}(s)$ to represent the control flow graph (CFG) of s. Formally, $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ is a collection of basic blocks $\mathcal{V}$ and directed edges between them $\mathcal{E}$. Each basic block $v \in \mathcal{V}$ may contain one or more instructions $i_1 i_2 \cdots i_n$. Each directed edge $e \in \mathcal{E}$ describes a *may-reachable* property between two basic blocks. For example, $e = (v_1, v_2) \in \mathcal{E}$ means basic block v_1 may reach v_2 in real execution.

Next, we explain the definition of smart contract *birthmark*. A birthmark is denoted as $\mathcal{M}$ and includes two types of metadata, *i.e.*, syntactic bytecode metadata ($\mathcal{M}_s$) and transaction sketch metadata ($\mathcal{M}_t$) respectively. Given a basic block v, a birthmark of v is a numeric tuple $\mathcal{M}(v) = \langle \mathcal{M}_s(v), \mathcal{M}_t(v) \rangle$. In practice, $\mathcal{M}(v)$ is used as an abstract representation of the basic block v. In terms of a smart contract with a CFG $\mathcal{G}(s) = \langle \mathcal{V}_s, \mathcal{E}_s \rangle$, $\mathcal{M}(s)$ is a collection of birthmarks with all its basic blocks combined, *i.e.*, $\mathcal{M}(s) = \{\mathcal{M}(v) \mid v \in \mathcal{V}_s\}$. The major strength of birthmark is the capability to enable straightforward computation on smart contracts (via vector calculation) and capture a good degree of high-level semantics as explained later.

In terms of the syntactic bytecode metadata, we consider six categories of statistics based on the specification of EVM [1]. Therefore, $\mathcal{M}_s$ is a 6-tuple vector $\langle s_a, s_l, s_e, s_b, s_s, s_m \rangle$, where each element is the number of instructions of the corresponding category, *i.e.*, arithmetic instructions (*e.g.*, `ADD`), logic instructions (*e.g.*, `AND`), environment instructions (*e.g.*, `BALANCE`), blockchain instructions (*e.g.*, `GASLIMIT`), stack instructions (*e.g.*, `PUSH`), memory instructions (*e.g.*, `MSTORE`). On the other hand, $\mathcal{M}_t$ aims at modeling the high-level semantics of smart contracts by generating transaction sketches at the basic block level. Specifically, a sketch $\mathcal{M}_t(v) = \langle C, P \rangle$ of basic block v

```
1  function transfer(address recv, uint amount) onlyOwner {
2    total = total + amount; // 2:[L,S,UU]
3    if(total > 1000) { // 3:[L] 2-3:[DU]
4      total = 0; // 4:[S] 3-4:[UU]
5    } else {
6      if(block.number % 2 == 0) {
7        recv.call.value(amount)(); // 2-7:[UpC,UsC] 3-7:[UsC]
8      }
9      last = recv; // 9:[S] 7-9:[CF]
10   }
11 }
```

(a) A Solidity function `transfer`



(b) Part of the `transfer` bytecode

Fig. 4: An illustrative example to explain the smart contract birthmark

consists of two types of metadata, *i.e.*, *path condition* (C) of v, *semantic properties* (P) of v. While C models the control flow dependency of v, P abstracts its high-level behavior via defined bytecode patterns of storage accesses and message calls (*i.e.*, state-changing operations for smart contracts [2]). For example, the branch at line 5 of Figure 1 creates a $c \in C$ that `balances[msg.sender]` $<$ `amount`. As for semantic properties, the defined patterns are described in Table I.

TABLE I: Defined semantic properties to model storage accesses and message calls. `o` and `v` are the storage offset and the value stored in the storage, respectively.

Property	Description
L	Single `SLOAD o` operation
S	Single `SSTORE o v` operation
C	Single `CALL` operation
DU	*Def-Use* pattern, *e.g.*, `SSTORE o v; SLOAD o;`
UU	*Use-Update* pattern, *e.g.*, `SLOAD o; SSTORE o v;`
UpC	*Update-Call* pattern, *e.g.*, `SSTORE o v; CALL;`
UsC	*Use-Call* pattern, *e.g.*, `SLOAD o; CALL;`
CF	*Call-Finalize* pattern, *e.g.*, `CALL; SSTORE o v;`

In Table I, we have defined two classes of semantic properties, *i.e.*, single-instruction property (top half) and cross-instruction property (bottom half). The first three properties (*L*, *S*, *C*) target at the instructions of `SLOAD`, `SSTORE` and `CALL`, which are often used to manipulate important storage data and communicate with external blockchain addresses. The rest five properties are designed to capture patterns involving two instructions. Although these patterns may not cover all the cases, they are widely implemented in Ethereum smart contracts, thus are closely related to the high-level programming intents of contract developers. Next, we use an example in Figure 4 to further explain the smart contract birthmark.

Illustrative Example. Figure 4a shows a Solidity function `transfer` which takes two arguments (*i.e.*, `recv` and `amount`) and operates on two storage data (*i.e.*, `total` and `last`). Particularly, a modifier `onlyOwner` is used to restrict function calls from non-owner addresses[1], *e.g.*, `require(msg.sender == owner)`. Figure 4b shows two bytecode snippets of the `transfer` function, which are compiled from the `onlyOwner` modifier (line 1) and branch (line 6) respectively. The syntactic bytecode metadata, *i.e.*, $\mathcal{M}_s$, is further explained in Figure 4b by classifying instructions into corresponding groups as aforementioned. For example, the `CALLER` instruction which puts the address of message caller onto stack is counted by s_e as defined in $\mathcal{M}_s$. Similarly, blockchain related instructions (*e.g.*, `NUMBER` that gets the number of the current block) are labeled as s_b. Next, we describe the symbolic sketch metadata $\mathcal{M}_t$, including path conditions and semantic properties. Although our approach works at bytecode level, we use the Solidity smart contract here for better explanation. Specifically, we use the message call operation at line 7 of Figure 4a as an example. This line of code has two path conditions from the entry of the `transfer` function, *i.e.*, `total <= 1000` and `block.number % 2 == 0`. Using an SMT solver (*e.g.*, Z3 [5]), the conditions may be modeled as `ULE(total,1000)` and `EQ(MOD(block.number,2),0)`, respectively. Furthermore, path conditions $C = \langle c, o \rangle$ are encoded using the number of constraints (*i.e.*, c, 2 in this case) and operators (*i.e.*, o, 3 in this case including `ULE`, `EQ` and `MOD`). In terms of the semantic properties as in Table I, we specified property instances in the comments of Figure 4a. For example, line 2 introduces an *L* property (load value from storage `total`), an *S* property (store value to `total`) and a *UU* property which combines *L* and *S*. Moreover, a *DU* property exists from line 2 to 3 by first defining `total` and then using it. Similarly, from line 2 to 7, *UpC* and *UsC* properties are modeled since a message call at line 7 follows a storage update and a usage at line 2. A *CF* property is lastly identified from 7 to 9 due to the finalize update at line 9 after the message call at line 7. The birthmark of the illustrative example transforms both $\mathcal{M}_s$ and $\mathcal{M}_t$ as aforementioned into numeric vectors.

B. Birthmark Generation

As introduced in §I, a birthmark is automatically generated from a given smart contract. This is realized via *symbolic transaction* (which is explained later), *i.e.*, symbolically execute a smart contract with symbolic blockchain values. Specifically, symbolic transaction works in two steps. When executing a basic block of a smart contract, it first parsed the bytecode in the block statically to retrieve syntactic bytecode metadata. Then, as the symbolic execution flows within the block, transaction sketch metadata is generated on the fly. As traditional symbolic execution techniques [6], [7], symbolic

[1] https://solidity.readthedocs.io/en/v0.4.24/common-patterns.html

transaction aims at covering as many program paths as possible. In our setting, birthmarks of basic blocks in uncovered paths only include syntactic bytecode information. Next, we describe the process of symbolic transaction and birthmark generation in detail.

Generally, symbolic transaction takes as input a CFG of the a smart contract and a set $\mathcal{I} = \langle \mathcal{I}_s, \mathcal{I}_a, \mathcal{I}_v, \mathcal{I}_d, \mathcal{I}_H \rangle$ of symbolic runtime parameters. Specifically, $\mathcal{I}_s$ is the address of the message sender. $\mathcal{I}_a$ is the receiver address of the message (the address of the smart contract in our case). $\mathcal{I}_v$ is the amount of ether attached in the transaction. $\mathcal{I}_d$ is the input data of the transaction. $\mathcal{I}_H$ is the header information [1], including coinbase, block number, difficulty value, gas limit *etc.*. To execute the transaction, a symbolic execution engine is used in our framework, whose responsibility is iteratively fetching a basic block from the CFG of the smart contract and then interpreting all the instructions within the block using symbolic parameters. Particularly, we focus on three types of instructions, *i.e.*, `SSTORE`, `SLOAD` and `CALL`. When executing an `SSTORE` instruction, the top two elements in EVM are an address o of the storage and a value v to store. We record o and attach it to the `SSTORE` instruction. For `SLOAD`, o is also recorded. In terms of `CALL`, the second top element indicates the address of the message recipient, which is associated with the instruction. Furthermore, before the symbolic transaction steps into a basic block, the path condition of this block is captured for later use. In practice, symbolic transaction traverses the CFG of a smart contract and label specific information to basic blocks and instructions as aforementioned. When the symbolic transaction terminates, we generate birthmarks for all the basic blocks in the CFG of the smart contract. Algorithm 1 explains the birthmark generation process for a single basic block.

The algorithm takes as input a basic block v and generates its birthmark $\mathcal{M}(v)$. Two state data structures, *i.e.*, a dictionary Q and a list k_q, are declared and initialized (line 5). Before the algorithm enters v, it parses the path condition expression and generates an tuple $\langle c, o \rangle$ as an embedding (line 6). Then, we generate $\mathcal{M}_s(V)$ and P by processing all the instructions in v based on their mnemonic (line 7 to 29). For `SSTORE` and `SLOAD`, we maintain a queue q to store a sequence of operations on a specific storage address *addr*. Then, the update of P is realized via parsing q and identifying semantic properties. Similarly, the generation of properties which are related to `CALL`, *i.e.*, *C*, *UpC* and *UsC*, is implemented via a traversal of k_q. For other types of instructions in v, the algorithm takes the current instruction and updates the corresponding element of $\mathcal{M}_s(v)$. Lastly, the birthmark of the basic block v is produced by combining $\mathcal{M}_s(v)$, C and P.

C. Clone Detection

Next, we describe our clone detection technique based on smart contract birthmarks. Generally, the detection is realized by computing the similarity of birthmarks between two contracts. Specifically, we first define the distance of two numerical vectors. Given two vectors P and Q, their distance $\|PQ\|$ is defined as in Formula Vector Distance.

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

Algorithm 1: Generate birthmark for a basic block

Input : v is the basic block to be executed.
Output: $\mathcal{M}(v)$ is birthmark of v.

1 $\mathcal{M}_s(v) = \langle s_a, s_l, s_e, s_b, s_s, s_m \rangle \leftarrow \langle 0, \cdots, 0 \rangle$
2 $\mathcal{M}_t(v) = \langle C, P \rangle \quad C = \langle c, o \rangle \leftarrow \langle 0, 0 \rangle$
3 $P = \langle \boldsymbol{L, S, C, DU, UU, UpC, UsC, CF} \rangle \leftarrow \langle 0, \cdots, 0 \rangle$
4 $\mathcal{I} \leftarrow$ `get_instructions`(v)
5 $Q \leftarrow \{\} \quad k_q \leftarrow [\;]$
6 $C \leftarrow$ `update_C`(`get_path_condition`(v))
7 **for** i **in** $\mathcal{I}$ **do**
8   **switch** *mnemonic*(i) **do**
9   *addr* $\leftarrow$ `get_storage`(i)
10         $q \leftarrow Q[addr]$
11   **case** `SSTORE` **do**
12   `update_S`(P, q) `// update S property`
13   `update_UU`(P, q) `// update UU property`
14   `update_CF`(P, q) `// update CF property`
15             $Q[addr] \leftarrow$ `append`(q, i)
16   `append`(k_q, i)
17   **case** `SLOAD` **do**
18   `update_L`(P, q) `// update L property`
19   `update_DU`(P, q) `// update DU property`
20             $Q[addr] \leftarrow$ `append`(q, i)
21   `append`(k_q, i)
22   **case** `CALL` **do**
23   **for** k, v**in** Q **do**
24                 $Q[k] \leftarrow$ `append`(v, i)
25   `update_C`(P) `// update C property`
26   `update_UpC`(P, k_q) `// update UpC property`
27   `update_UsC`(P, k_q) `// update UsC property`
28   **otherwise do**
29   `update_Ms`$(\mathcal{M}_s(v), i)$ `// update` $\mathcal{M}_s(v)$
30 **return** $\mathcal{M}(v) \leftarrow \langle \mathcal{M}_s(v), C, P \rangle$

We adopt a similar distance definition as in [8]. Intuitively, similar vectors are guaranteed to produce a low distance, and vice versa. Particularly, α_i is a set of parameters to indicate the relative significance of different fields in a vector, *i.e.*, which field has bigger impact on identifying the similarity of two vectors. In practice, α_i can be automatically inferred via supervised learning. For birthmarks $\mathcal{M}(v_1) = \langle \mathcal{M}_s(v_1), \mathcal{M}_t(v_1) \rangle$ and $\mathcal{M}(v_2) = \langle \mathcal{M}_s(v_2), \mathcal{M}_t(v_2) \rangle$ of basic blocks v_1 and v_2, we can use Formula Vector Distance to compute the distance between syntactic bytecode metadata (denoted as $\|v_1 v_2\|_s = \|\mathcal{M}_s(v_1) \mathcal{M}_s(v_2)\|$) and transaction sketch metadata (denoted as $\|v_1 v_2\|_t = \|\mathcal{M}_t(v_1) \mathcal{M}_t(v_2)\|$) of v_1 and v_2, respectively. Then, we further define the similarity $\|v_1 v_2\|$ of two basic blocks, *e.g.*, v_1 and v_2, as in Formula Block Similarity.

$$\|v_1 v_2\| = \frac{1 - \|v_1 v_2\|_t}{1 + \|v_1 v_2\|_t + \omega \cdot \|v_1 v_2\|_s} \qquad \text{(Block Similarity)}$$

In this formula, ω is a parameter commonly with a small value for tuning the weight of syntactic bytecode metadata in the case of clone detection. That said, the similarity between two basic blocks is mainly determined by their transaction sketch metadata, *i.e.*, how similar the blocks behave in the same transaction. We design the block similarity in this way so that the clone detection can better capture high-level programming intents without being mis-guided by syntax noise. Based on the block similarity, we use $P(v_1; v_2)$ to denote the probability measurement that v_1 and v_2 are basic block level semantic clones, *i.e.*, v_1 is semantically equivalent or similar to v_2. The probability is computed as in Formula Clone Blocks.

$$P(v_1; v_2) = 1/(1 + e^{-k\cdot(\|v_1 v_2\| - 0.5)}) \qquad \text{(Clone Blocks)}$$

The probability is estimated by applying a sigmoid function with a midpoint to be 0.5 as $\|v_1 v_2\| \in [0, 1]$ [9]. Next, we describe the contract level similarity based on clone blocks. Assuming $\mathcal{G}_1$ and $\mathcal{G}_2$ are two CFG from a pair of smart contracts, the basic idea of computing similarity between $\mathcal{G}_1$ and $\mathcal{G}_2$ is to find the *best match* (*i.e.*, biggest probability measurement between two blocks) in $\mathcal{G}_2$ for each basic block in $\mathcal{G}_1$, and vice versa. Moreover, the task of searching best matches is implemented as identifying a pair of basic blocks with a smallest vector distance for transaction sketch metadata, without syntactic bytecode metadata involved. In this way, we can avoid a portion of false matches whose similarity is not mainly contributed by semantic information, *e.g.*, path conditions, storage accesses and message calls *etc.*, but other less important instructions. When a pair of matched blocks is discovered, we compute their clone probability measurement (as in Formula Clone Blocks). Lastly, we define an asymmetric clone probability $\mathit{Sim}(s_1 \to s_2)$ from contract s_1 to s_2 via the Clone Prob Formula.

$$\mathit{Sim}(s_1 \to s_2) = \sum_{v_i \in s_1} \log \frac{P(v_i; v^*)}{P(v_i; H_0)} \qquad \text{(Clone Prob)}$$

In particular, given a specific basic block $v_i \in s_1$, $v_j \in s_2$ and $v^* = \text{argmax}\|v_i v_j\|_t$. $P(v_i; H_0)$ represents a probability estimation of clones between v_i and a random basic block H_0. A simple method to estimate H_0 is to use an average similarity between v_i and all the basic blocks in s_2. The absolute similarity between s_1 and s_2 is denoted as $\mathit{Sim}(s_1, s_2)$ and computed by $\max\{\mathit{Sim}(s_1 \to s_2), \mathit{Sim}(s_2 \to s_1)\}$. We further use the relative similarity to detect contract clones as defined in Clone Contracts Formula.

$$\mathit{Sim}^*(s_1, s_2) = \mathit{Sim}(s_1, s_2)/\mathit{Sim}(s_2, s_2) \qquad \text{(Clone Contracts)}$$

Given a threshold ϕ, s_1 and s_2 are considered as clones if $\mathit{Sim}^*(s_1, s_2) \geq 1 - \phi$. Otherwise, they are marked as unrelated ones *w.r.t.* ϕ.

IV. Empirical Evaluation

A. Experiment Setup

Implementation. We have developed a clone detector called *EClone* for Ethereum smart contracts. Specifically, *EClone* leverages Oyente [10] to construct CFG from EVM bytecode and perform symbolic transaction. Moreover, we implemented the modules of metadata extraction, similarity computation and clone detection in Python. Additionally, *EClone* uses a training module to train and optimize the parameters used in the clone detection, *e.g.*, α, ω, k as mentioned in §III. To this end, we prepare a corpus $\mathcal{C}$ 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 following Objective Function.

$$\max \sum (\mathit{Sim}^*(c_i, c_j) - \mathit{Sim}^*(c_i, u_k)) \qquad \text{(Objective Function)}$$

Specifically, c_i and c_j are labeled as 1 and c_i and u_k are labeled as -1 in the training data ($c_i, c_j, u_k \in \mathcal{C}$). Conceptually, the objective function helps *EClone* separate clone and unrelated contracts as much as possible. *EClone* is publicly available at `URL omitted for double-blind review`.

Dataset Preparation. All the experiments were performed on a Ubuntu 16.04 virtual machine with dual Intel Core i5 processors, 10GB RAM and 128GB SSD. We collected two types of evaluation corpus from Mainnet Etherscan [11], *i.e.*, $\mathcal{T}_{optimize}$ and $\mathcal{T}_{Dapp}$ respectively. The $\mathcal{T}_{optimize}$ consists of smart contract bytecode generated with and without `solc` compiler optimizations (enabled by the `--optimize` option) [2]. On the other hand, $\mathcal{T}_{Dapp}$ is produced by picking contracts from different Dapp domains, *i.e.*, Token, Lottery, Voting. $\mathcal{T}_{optimize}$ includes 3,156 test cases and $\mathcal{T}_{Dapp}$ contains 300 test cases. Specifically, each test case is a triple of $\langle q, t, l\rangle$, where q and t is a pair of EVM bytecode for clone detection. $l \in \{-1, 1\}$ is a label that indicates whether q and t are clones (1) or not (-1). For $\mathcal{T}_{optimize}$ corpus, we set a label to be 1 if q and t are the unoptimized and optimized versions of the same smart contract. Otherwise, we set the label to be -1. For $\mathcal{T}_{Dapp}$ corpus, a label is specified as 1 if q and t are from the same Dapp domain, and -1 if q and t come from different Dapp domains. To determine whether the birthmark-based clone detection is correct or not, we first used *EClone* to compute a label l' for q and t and then compared l' with the ground truth label l. Cases where $l = l'$ were considered as correct detections and $l \neq l'$ situations were regarded as false reports.

B. Research Questions

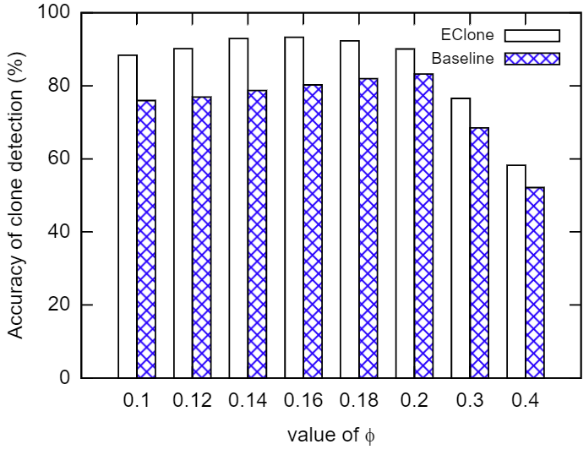
Question 1. Can *EClone* detect smart contract clones?
Question 2. Is transaction sketch metadata necessary?
Question 3. What are the practical values of *EClone*?

C. Empirical Results

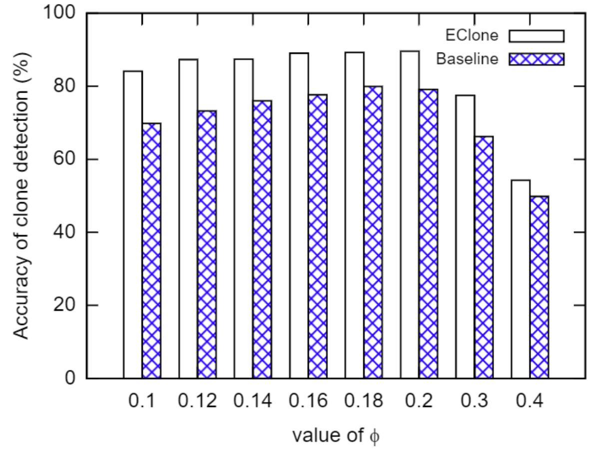
To answer the aforementioned questions in the setting of Ethereum, we have conducted a set of empirical case studies on the task of smart contract clone detection using *EClone*. Next, we describe and explain the empirical results. In the first study, we used *EClone* to detect clones in smart contracts of $\mathcal{T}_{optmize}$. The detection results are shown in Figure 5a.

In the evaluation, we considered four types of statistics, *i.e.*, true positive (TP), true negative (TN), false positive (FP) and false negative (FN) respectively. For example, if the label l of the test case is 1 and *EClone* also generates 1 label, we count this test case as TP. However, if *EClone* generates -1, an FN will be recorded. In Figure 5a, we computed the precision of *EClone* under different values of the detection threshold ϕ. Specifically, the precision is calculated as $\frac{TP+TN}{N}$ where

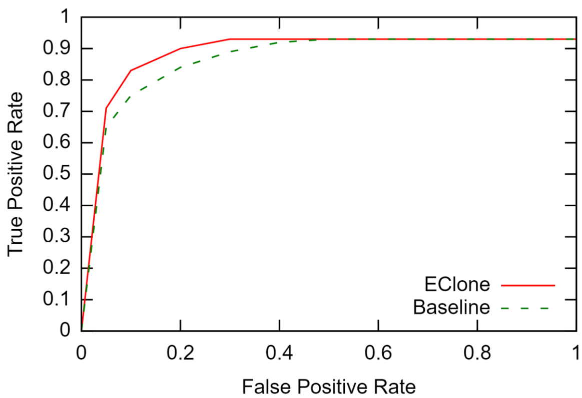
[2]https://github.com/gorkazl/pyGAlib



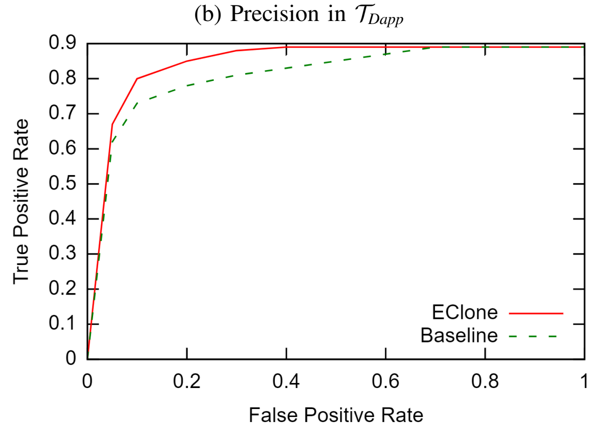
(a) Precision in $\mathcal{T}_{optimize}$



(b) Precision in $\mathcal{T}_{Dapp}$



(c) ROC in $\mathcal{T}_{optimize}$



(d) ROC in $\mathcal{T}_{Dapp}$

Fig. 5: Clone detection results in $\mathcal{T}_{optimize}$ and $\mathcal{T}_{Dapp}$. Specifically, (a), (b), (c), (d) are precision and ROC curves compared to the baseline approach. (a) and (c) are for $\mathcal{T}_{optimize}$ while (b) and (d) are for $\mathcal{T}_{Dapp}$.

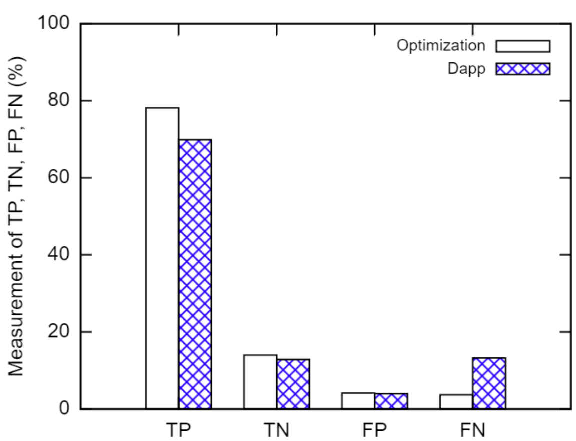


Fig. 6: Detection difference between $\mathcal{T}_{optimize}$ and $\mathcal{T}_{Dapp}$.

$N = TP + TN + FP + FN$. The precision measurement indicates the capability of a clone detector to not only find similar code but filter irrelevant ones as well. In the first case study, we used 8 different values of ϕ (from 0.1 to 0.4) to perform clone detection. Figure 5a showed that *EClone* achieved a precision from 58.21% to 93.27%, where $\phi = 0.16$ outperformed other settings. While bigger thresholds introduced false positives, smaller thresholds led to false negatives. In practice, specifying an optimal threshold for clone detection is essential, but unfortunately intractable. We suggested picking a ϕ empirically based on a specific application setting. In terms of clones derived from compilation optimizations, Figure 5a highlighted a strategy to configure the threshold. On the other hand, we analyzed the precision of clone detection achieved by *EClone* in the setting of cross-domain Dapp, as described in Figure 5b. Given different values of detection threshold ϕ, the precision of *EClone* ranges from 54.23% to 89.57%. Although the a minor decrease on precision was observed compared to the setting of detecting clones against compiler optimizations (*i.e.*, $\mathcal{T}_{optimize}$), the results still demonstrated a potential of *EClone* in finding real smart contract clones, thus can help answer **Question 1** in affirmative.

Furthermore, we investigated the difference manifested between clone detections in $\mathcal{T}_{optimize}$ and $\mathcal{T}_{Dapp}$. Specifically, we compared the two types of test cases over TP, TN, FP, FN measurements, as shown in Figure 6. From the results, it is straightforward to see that while TN and FP measurements were close for both types of test cases, detection results of $\mathcal{T}_{Dapp}$ displayed higher FN and lower TP, thus a relatively lower precision. The difference of precision can be further

```
if(balances[msg.sender]>=v &&
    balances[to]+v >= balances[to]){
   balances[msg.sender]-=v;
   balances[_to]+=v;
   Transfer(msg.sender, to, v);
   return true;
} else {
   return false;
}
```

(a) HBToken

```
if (to == 0x0)
  throw;
if (v <= 0)
  throw;
if (balanceOf[msg.sender] < _v)
  throw;
if(balanceOf[to]+v < balanceOf[to])
  throw;
balances[msg.sender] -= v;
balances[to] += v;
Transfer(msg.sender, to, v);
```

(b) BNB Token

```
claimBonus(msg.sender);
claimBonus(to);
if(balences[msg.sender] >= v &&
   v > 0) {
   balences[msg.sender] -= v;
   balences[to] += v;
   Transfer(msg.sender, to, v);
   return true;
} else {
   return false;
}
```

(c) VEN Token

Fig. 7: Three `transfer(address _to, uint _v)` function variations in $\mathcal{T}_{Dapp}$.

explained by the difference of EVM bytecode from the two groups of smart contracts. Compared to the bytecode variations introduced by compiler optimization in $\mathcal{T}_{optimize}$, $\mathcal{T}_{Dapp}$ incurred more diversity for smart contracts even within the same application domain. To better explain the diversity for $\mathcal{T}_{Dapp}$ test cases, we listed three representative Solidity functions of Token contracts in Figure 7.

The three variations of token smart contracts all declared a `transfer` function, which is an interface from `ERC20` token standard [12] and takes as input a recipient address `_to` and the amount of tokens `_v` for the transfer. As a result, the three variations have a similar function structure, *i.e.*, sanity checks on whether the sender has enough tokens and the receiver can correctly take the transfer (without overflow) followed by a sequence of actual token transfer from the sender to the receiver. Particularly, `BNB` token in Figure 7b defines extra checks to validate the address of the recipient and the value associated to the transfer. In the context of clone detection, *EClone* can detect the similarity between Figure 7a and Figure 7b despite the code differences. The detection was realized by effectively identifying important patterns which indicate high-level intents. For example, the accesses on the storage `balences` are similar across the three variations, which further generates similar semantic properties for *EClone*. However, the `VEN` function in Figure 7c specified a different implementation which introduced a new strategy in the process of token transfer, *i.e.*, `claimBonus` function call, as shown in the first two lines in Figure 7c. Consequently, the `VEN` contract included the body of the `claimBonus` function, which caused a major difference to the other two contracts. In this case, *EClone* failed to identify such clones in $\mathcal{T}_{Dapp}$ and produced a false negative (FN).

Furthermore, we conducted a comparison experiment between *EClone* and a baseline approach. Specifically, the baseline used only syntactic features for clone detection. That is, transaction sketch metadata was removed from the birthmark. Instead, instructions of storage accesses (`SLOAD`, `SSTORE`) and message calls (`CALL`) were counted and combined with the syntactic bytecode metadata for a single basic block. Then a graph matching algorithm was performed based on subgraph isomorphism [8]. The results are shown in Figure 5. Compared to the baseline, *EClone* achieved a better precision of clone detection *w.r.t.* both $\mathcal{T}_{optimize}$ and $\mathcal{T}_{Dapp}$ settings. The optimization over baseline was 12.09% for $\mathcal{T}_{optimize}$ and 12.12% for $\mathcal{T}_{Dapp}$ respectively, as in Figure 5a and 5b. Moreover, we have computed the ROC curves in these two settings, as shown in Figure 5c and 5d. From the results, it is clear to see that given the same false positive rate, *EClone* generated a higher true positive rate than the baseline. On the other hand, under the same true positive rate, *EClone* manifested a lower false positive rate than the baseline. That said, *EClone* was a better fit in the context of detecting smart contract clones compared to the baseline approach. Considering the technical difference between *EClone* and baseline, the achieved optimization indicated the awareness of semantics introduced by the transaction sketch metadata, *i.e.*, an essential form of information provided by *EClone*. Specifically, the sketch metadata facilitated the identification of high-level programming intents via a set of predefined types of semantic properties (as in §III-A) at basic block level, and further enhanced the clone detector. Therefore, we can respond to **Question 2** positively.

D. Application of EClone: Vulnerability Search

We have further instantiated an application of *EClone*, *i.e.*, vulnerability search. That is, given a target vulnerable function t and a set of contracts C, we detect variants of t in $c \in C$. From the practical perspective, vulnerability search for smart contracts is important to secure blockchain ecosystems. Taking the DAO attack [13] as an example, vulnerability search could enable us to very quickly find DAO-like problems in other contracts before they got exploited. Moreover, designing detection patterns for security problems can sometimes be difficult or even impossible (*e.g.*, harmful integer overflows). In such cases, the capability of vulnerability search can help effectively catch known issues even if we do not have a precise analysis algorithm.

Extension of EClone. We extended *EClone* to search for function-level vulnerabilities. Based on the structure of EVM bytecode, a contract CFG is organized as a "function selector" followed by a set of function subgraphs. For a function `foo`, the selector uses `PUSH4 hash(foo)` to put hash of `foo` onto stack. Then, it extracts the first four bytes from transaction data and compares the bytes with `hash(foo)`. If the two match, then the execution goes to `foo`. Using this heuristic, we recognize function hashes for basic blocks when performing symbolic transaction. Given a vulnerable contract c_t, the hash of the vulnerable function h and a contract c_q to be searched, *EClone* extracts a set $c_t(h)$ basic blocks from c_t based on h. Similarly, we extracted $c_q(h_0)c_q(h_1)\cdots c_q(h_m)$ from c_q where $h_0h_1\cdots h_m$ are function hashes in c_q. Lastly, *EClone* generates a ranking on these functions based on $Sim^*(c_q(h_i), c_t(h))$, where larger values indicate higher probabilities to have the same vulnerability.

Case Study Setup. In our case study, we applied *EClone* to search for the `CVE-2018-10376` vulnerability of smart contracts. The vulnerable target function `transferProxy` at `3ac6cb00f5a44712022a51fbace4c7497f56ee31` is shown in Figure 8. Specifically, the contract sets up a proxy for users to transfer cryptocurrencies between accounts. However, due to an unprotected integer overflow (*i.e.*, `_fee + _value`) at line 5, an attacker could bypass the sanity check at line 6 and transfer a large amount of money to specific accounts (line 13 and 15) from a zero-balance account (line 17). In the study, we have collected a group of five real-world contracts in Ethereum which are affected by `CVE-2018-10376` [14], as shown in Table II. Specifically, some of the contracts were quite active with more than 50,000 transactions in total, which highlighted the significance of finding the vulnerability in time.

```
1  function transferProxy(address _from, address _to,
2    uint256 _value, uint256 _fee, uint8 _v,bytes32 _r,
3    bytes32 _s) public transferAllowed(_from)
4    returns (bool) {
5    if(balances[_from] < _fee + _value)
6      revert();
7    uint256 nonce = nonces[_from];
8    bytes32 h = keccak256(_from,_to,_value,_fee,nonce);
9    if(_from != ecrecover(h,_v,_r,_s)) revert();
10   if(balances[_to] + _value < balances[_to] ||
11   balances[msg.sender]+_fee<balances[msg.sender])
12   revert();
13   balances[_to] += _value;
14   Transfer(_from, _to, _value);
15   balances[msg.sender] += _fee;
16   Transfer(_from, msg.sender, _fee);
17   balances[_from] -= _value + _fee;
18   nonces[_from] = nonce + 1;
19   return true;
20 }
```

Fig. 8: `CVE-2018-10376`

TABLE II: Contracts affected by `CVE-2018-10376`.

Contract	Address	Tx
SMART	60be37dacb94748a12208a7ff298f6112365e31f	643
MTC	8febf7551eea6ce499f96537ae0e2075c5a7301a	4,121
MESH	01f2acf2914860331c1cb1a9acecda7475e06af8	10,289
UGToken	43ee79e379e7b78d871100ed696e803e7893b644	42,882
SMT	55F93985431Fc9304077687a35A1BA103dC1e081	54,545

Vulnerability Ranking. Next, we used the extended *EClone* to rank functions in the five vulnerable contracts based on how similar the function is to `CVE-2018-10376`. For each contract, two versions of bytecode were used, *i.e.*, un-optimized and optimized ones. The ranking is shown in Table III. For un-optimized contract bytecode of `MESH`, `SMT`, `MTC` and `SMART`, *EClone* ranked the vulnerable function as top one *w.r.t.* around 25 functions in total. For un-optimized `UGToken` contract, the vulnerable function was the second in the ranking. In terms of

TABLE III: Function ranking of smart contracts affected by `CVE-2018-10376`. **#Basic Block** and **#Function** columns show the number of basic blocks and functions in the contracts. **No-opt** and **Opt** columns are results of contract bytecode generated without and with compiler optimization respectively.

Contract	#Basic Block	#Function	No-opt	Opt
MESH	321	27	1	1
SMT	293	26	1	1
MTC	247	23	1	1
SMART	235	22	1	2
UGToken	211	19	2	2

optimized contracts, *EClone* managed to identify vulnerable functions in the top two of the ranking list. In this sense, the process of vulnerability search can be more efficient by focusing on only the top functions generated by *EClone*.

TABLE IV: Comparison on vulnerability detection of `CVE-2018-10376`. Sim means relative similarity between the vulnerable function in these contracts and the target in Figure 8. **Oyente** was configured with different timeout for SMT solving, *i.e.*, 100ms and 1000ms respectively.

Contract	EClone		Oyente		Securify
	Sim	$\phi = 0.16$	100ms	1000ms	
MESH	0.96	✓	×	×	×
SMT	0.99	✓	×	×	×
MTC	0.95	✓	×	×	×
SMART	0.98	✓	×	×	×
UGToken	0.79	×	×	×	×

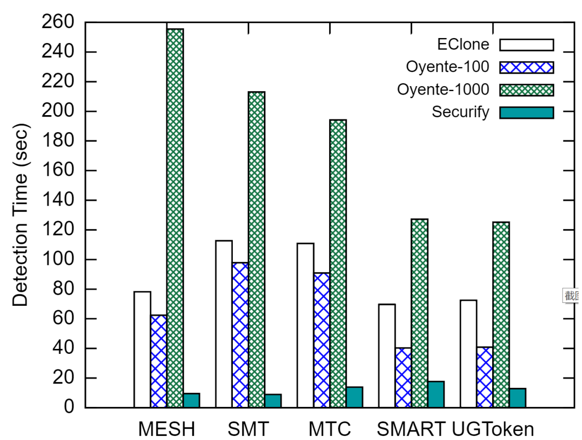


Fig. 9: Performance of vulnerability detection.

Vulnerability Detection. Furthermore, we have compared *EClone* with two existing security analyzers of smart contracts, *i.e.*, Oyente [10] and Securify [15], in terms of vulnerability detection. These two analyzers are based on symbolic execution and formal verification techniques. Specifically, we used two configurations of Oyente with different timeout for SMT solving, *i.e.*, 100ms and 1000ms respectively. For *EClone*, we set $\phi = 0.16$ as the threshold to flag potential vulnerabilities. The comparison is explained as in Table IV. While *EClone* successfully found the vulnerable functions in four contracts with relatively high confidence (relative similarities against `CVE-2018-10376` were greater than 0.95), both Oyente and

Securify failed to report the vulnerability that was caused by a simple integer overflow bug. The main reason for this missed bug is that the logic in `CVE-2018-10376` (line 5, 6) is similar to a widely used pattern for overflow protection, *i.e.*, addition overflow followed by a `revert`. In this case, the analyzers mistakenly identified the bug as a protection pattern thus led to a false negative. On the other hand, *EClone* detected this vulnerability via analyzing the similarity to the target and avoided getting confused by specific code patterns. In general, strengths of such in-depth analyzers (*e.g.*, Oyente and Securify) and *EClone* are orthogonal to each other. Well-designed combinations could create stronger security analysis capability for both sides. In this sense, we can respond to **Question 3** that *EClone* highlighted a practical way to detect security threats in Ethereum contracts, as a compliment to existing solutions. Lastly, we analyzed the runtime overhead of *EClone* in finding `CVE-2018-10376`, as in Figure 9. Compared to Oyente with 100ms SMT solving, *EClone* was slower by around 40% on average due to birthmark generation and similarity computation. With most of the overhead coming from computations on independent vectors, hardware-supported parallelization in *EClone* might be possible and is left for future work.

V. Related Works

Software Birthmarks. A birthmark of software was initially designed to identify intrinsic software properties and fight against code theft [16]–[18]. Myles and Collberg introduced *whole program path birthmark*, which models a program via a complete control flow trace of its execution [16]. Tamada *et al.* proposed to observe the runtime interaction between an application and its environment [17]. In the context of Java applications, Schuler *et al.* presented a new type of birthmark by monitoring how objects are used via Java Standard API [18]. In order to capture unique behavior in large-scale software, Wang *et al.* designed the birthmark of *system call dependence graph* to encode programs via their calls to the operating system [19]. In the context of blockchain, we designed the first birthmark for smart contracts. Particularly, we focus on critical semantic properties at runtime rather than the complete execution trace, thus can reduce complexity and capture programming intents as well.

Clone Detection. The topic of clone detection has been attracting research interests in the field of software engineering for a long time. Previous works focused on finding clones in both source code [20]–[22] and binary code [8], [9], [23]–[25]. Jiang *et al.* proposed the Deckard algorithm to identify similar tree representations of source code via clustering numeric vectors generated from subtrees [20], which is robust against minor code modifications and can scale for large programs. Gabel *et al.* extended Deckard by mapping program dependency graphs to their associating AST forests [21], such that the computation of graph comparison is reduced to a tree similarity problem. In the context of binary code, Saebjornsen *et al.* followed the original idea described in Deckard and extended it by normalizing assembly instructions with essential structure information considered [23]. David *et al.* further introduced input-output equivalence to check semantic similarity [9]. The equivalence is analyzed as a model checking problem thus is more robust against binary transformations such as obfuscation and optimizations. Chandramohan *et al.* applied a selective inlining technique on library and user-defined functions in detecting similar code [26]. In order to speed up the detection, Eschweiler *et al.* proposed numeric and structural filters to quickly identify unrelated programs [8]. Moreover, Xu *et al.* designed a neural network based approach to compute an embedding for binary functions. Compared to other graph-matching algorithms, this approach is more efficient and can flexibly adapt in various settings. Based on deep learning techniques, Liu *et al.* proposed to find similar binary based on intra-function, inter-function and inter-module features, which are directly generated from raw bytes rather than syntactic information as CFG [27]. Unlike clone detection methods in the literature, we consider both syntactic features and observable behavior as well. Furthermore, we modeled unique semantics of Ethereum such as storage accesses and inter-contract message calls and highlighted the first clone detector for blockchain applications.

Smart Contract Analysis. As blockchain has been gathering an increasing popularity recently, smart contract analysis has attracted more and more research interests across various topics, especially for the security issue of Ethereum. Luu *et al.* proposed Oyente, a symbolic executor for EVM bytecode. They defined four types of smart contract bugs and corresponding detections [10]. Kalra *et al.* designed Zeus which leveraged abstract interpretation and model checking to validate the fairness of smart contracts [28]. Tsankov *et al.* presented an automatic analyzer Securify for Ethereum, which extracts facts from contract code and verify the satisfaction of security properties [15]. Liu *et al.* focused on reentrancy bugs in smart contracts and leveraged fuzz testing to find them [29]. Based on the *n-gram* language model, Liu *et al.* proposed S-gram to identify statistical-abnormal code as potentially buggy contracts [30]. Furthermore, Krupp *et al.* introduced teTHER to generate exploits automatically for vulnerable contracts [31]. In our setting, we showed that clone detection can enable important applications for smart contracts, including vulnerability search on blockchain.

VI. Conclusion

In this paper, we have introduced smart contract birthmark for Ethereum. Specifically, a birthmark is an abstraction of EVM bytecode that captures high-level programming intents and automatically generated via symbolically executing transactions on a specific contract. Furthermore, we highlighted the application of clone detection in the context of smart contracts for the first time and showed that birthmark can be used as a good fit in this setting by enabling an easy way to similarity computation and alleviating the diversity of bytecode as well. We have also implemented a clone detector called *EClone* and evaluated it in finding real-world clones on Ethereum. The results demonstrated the effectiveness of *EClone* in accurately identifying clones across various compilation levels and Dapp domains as well. Moreover, we instantiated the application of vulnerability search using *EClone* and managed to detect `CVE-2018-10376` instances in different token contracts.

References

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, 2014.

[2] Ethereum, "Solidity — solidity 0.4.19 documentation," 2017. [Online]. Available: https://solidity.readthedocs.io/en/develop/
[3] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 837–847.
[4] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, 2014.
[5] "Microsoft z3 smt solver," https://z3.codeplex.com/, 2017.
[6] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.
[7] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: A software testing service," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 5–10, 2010.
[8] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code." in *NDSS*, 2016.
[9] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 266–280, 2016.
[10] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 254–269.
[11] Etherscan, "Etherscan," 2018. [Online]. Available: https://etherscan.io/
[12] Ethereum, "Ethereum," 2018. [Online]. Available: https://www.ethereum.org
[13] E. Blog, "Hard fork completed," 2016. [Online]. Available: https://blog.ethereum.org/2016/07/20/hard-fork-completed/
[14] Peckshield, "Peckshield," 2018. [Online]. Available: https://blog.peckshield.com/2018/04/25/proxyOverflow/
[15] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," *arXiv preprint arXiv:1806.01143*, 2018.
[16] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," in *International Conference on Information Security*. Springer, 2004, pp. 404–415.
[17] H. Tamada, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Design and evaluation of birthmarks for detecting theft of java programs." in *IASTED Conf. on Software Engineering*, 2004, pp. 569–574.
[18] D. Schuler, V. Dallmeier, and C. Lindig, "A dynamic birthmark for java," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 274–283.
[19] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 280–290.
[20] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.
[21] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 321–330.
[22] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: A tool for finding copy-paste and related bugs in operating system code." in *OSdi*, vol. 4, no. 19, 2004, pp. 289–302.
[23] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 117–128.
[24] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 480–491.
[25] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 363–376.
[26] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 678–689.
[27] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 667–678.
[28] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." NDSS, 2018.
[29] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *ICSE (Companion)*. ACM, 2018, pp. 65–68.
[30] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun, "S-gram: towards semantic-aware security auditing for ethereum smart contracts," in *ASE*. ACM, 2018, pp. 814–819.
[31] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*. USENIX Association, 2018, pp. 1317–1333.