A Large-Scale Empirical Study on Control Flow Identification of Smart Contracts

Ting Chen
Center for Cybersecurity
University of Electronic Science and Technology of China (UESTC)
Chengdu, China
brokendragon@uestc.edu.cn

Zihao Li
Center for Cybersecurity
UESTC
Chengdu, China
gforiq@qq.com

Yufei Zhang
Center for Cybersecurity
UESTC
Chengdu, China
2235285714@qq.com

Xiapu Luo*
Department of Computing
The Hong Kong Polytechnic University
Kowloon, Hong Kong
daniel.xiapu.luo@polyu.edu.hk

Ting Wang
Pennsylvania State University
Pennsylvania, USA
inbox.ting@gmail.com

Teng Hu
Center for Cybersecurity
UESTC
Chengdu, China
mailhuteng@gmail.com

Xiuzhuo Xiao
Center for Cybersecurity
UESTC
Chengdu, China
1015981305@qq.com

Dong Wang
ADLab of Venustech
Chengdu, China
wangdong2@venustech.com.cn

Jin Huang
ADLab of Venustech
Chengdu, China
huangjin2@venustech.com.cn

Xiaosong Zhang
Center for Cybersecurity
UESTC
Chengdu, China
johnsonzxs@uestc.edu.cn

Abstract*—Background: Millions of smart contracts have been deployed to Ethereum for providing various applications. Recent studies discovered many severe security and performance issues in smart contracts by applying static program analysis techniques to them. Given a smart contract, the majority of these analysis techniques need to first construct its control flow graph, which connects basic blocks through control flow transfers (CFTs), before conducting further analysis. Aims: The objective of this work is to understand the capabilities of static program analysis techniques to identify CFTs, and to investigate how static program analysis techniques can be improved if the CFTs are complemented. Method: We perform a comprehensive empirical study on six widely-used tools for smart contract analysis by using *all* deployed smart contracts to understand their capabilities to recognize CFTs. We capture *all* execution traces of all smart contracts to evaluate the number of CFTs covered by traces that are not found by those tools. We enhance a state-of-the-art tool, `OYENTE` for discovering vulnerabilities in smart contracts with the CFTs covered by traces to investigate how the tool is improved. Results: These studied tools fail to identify all CFTs due to several reasons, e.g., incomplete code patterns. Execution traces effectively complement these tool in recognizing CFTs. By including the CFTs covered by traces, the false negative rate of `OYENTE` can be reduced by up to 30%. Conclusions: Our study underlines the ineffectiveness of static analysis techniques due to the incapabilities of CFT identification.

I. Introduction

Ethereum is the most popular and largest blockchain platform supporting smart contracts. Millions of smart contracts have been deployed on Ethereum, and the number increases every day. A *smart contract* is a piece of autonomous *program* that executes the predefined logic automatically and mandatorily [1]. Smart contracts are usually developed in high-level programming languages (e.g., `Solidity` [2]) and then compiled into EVM (Ethereum Virtual Machine) bytecode, which will be executed by the EVM in every Ethereum node. Similar to programs developed in other languages for running in diverse devices, smart contracts are prone to flaws, which can cause severe financial losses. For example, a vulnerability in the DAO contract leads to 60 million USD loss [3], and a vulnerable contract developed by Parity inc. freezes 100 million USD so that users cannot withdraw money from it [4].

Recent studies have applied program analysis techniques to discover security and performance issues in smart contracts [5]–[18]. To name a few, `OYENTE` uses symbolic execution (SE) to reveal four kinds of vulnerable contracts [5]. `GASPER` detects three gas-inefficient code patterns (which will waste money of developers and users) in smart contracts by leveraging SE [11]. `SmartCheck` defines 21 bug patterns and looks for them in smart contracts [12]. `ContractFuzzer` applies fuzzing to smart contracts to discover seven kinds of security problems [17]. `Osiris` combines SE and taint analysis to discover integer overflow vulnerabilities in smart contracts [18]. `teEther` not only discovers security bugs in smart contracts, but also generates exploits [16].

Given a smart contract, the majority of these analysis techniques need to first construct its control flow graph (CFG), which connects basic blocks (BBs) through control flow transfers (CFTs), before conducting further analysis, because CFG represents the execution flow. A CFT from instruction A to B means that right after executing A, B will be executed. Identifying all CFTs is a necessary step to construct a complete and precise CFG. Since many security and reliability checking relies on CFGs, incomplete and/or imprecise CFGs will impair

* The corresponding author.

978-1-7281-2968-6/19/$31.00 ©2019 IEEE

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on August 12,2020 at 13:43:26 UTC from IEEE Xplore. Restrictions apply.

such checking. For instance, control flow integrity protection may raise alarms if not all valid CFTs are recognized. Besides, model checking may bias the results if the CFG does not faithfully represent the analyzed program.

Constructing accurate CFG is also important to smart contract analysis. For instance, OYENTE uses CFG to direct path exploration, and it discards a program path if the target of a jump operation cannot be determined. GASPER relies on CFGs to detect dead BBs, opaque predicates and discover loops [11]. In particular, it will miss a loop if it cannot identify the CFT corresponding to the back edge of a loop. As another example, sCompile, an SE-based tool, first constructs the CFG for a contract, and then identifies critical program paths from the CFG [15]. By doing so, sCompile can alleviate path explosion by just exploring critical paths, while leaving the other paths unexplored. Unfortunately, little is known about the capabilities of existing techniques for constructing CFGs of smart contracts, especially identifying CFTs, and the impact of inaccurate CFT identification.

To fill this gap, in this paper, we conduct the *first* in-depth analysis of CFT identification on smart contracts. Since the identification of CFTs highly depends on the instruction set, existing binary analysis tools [19]–[23] cannot handle smart contracts and the lessons learned from such tools cannot be directly applied to smart contracts. For example, in x86/x64, a CFT is usually caused by a JUMP, a CALL, or a RET instruction. In smart contracts, the CALL operation is used for inter-contract invocation (i.e., a contract calls the function of another contract) while intra-contract invocation (i.e., a function calls another function within the same contract) uses the JUMP operation. Moreover, in smart contracts, the target of a jump operation is stored in the runtime stack, rather than being encoded in the jump operation. Therefore, the runtime information is required to identify CFT, especially when the jump target is influenced by inputs. Differently, for binaries (e.g., x86), the target of a direct jump is encoded in the JUMP operation, which can be easily obtained.

To obtain convincing and representative observations, we perform a comprehensive examination on six widely-used tools [5]–[10] for smart contract analysis by using *all* deployed smart contracts from the launching of Ethereum to the preparation of this paper (~5 million smart contracts) and *all* their execution traces (60+ million execution traces), which are collected by strategically instrumenting the EVM. We run the selected tools to identify CFTs of these contracts, and contrast the CFTs discovered by the six tools and the CFTs extracted from *all* real execution traces. By quantitatively analyzing and comparing the results of these tools, we obtain many useful observations and insights. For example, OYENTE discovers more CFTs than the other SE-based tools while Porosity finds the most CFTs. Besides statistical results, we find several technical challenges that prevent existing techniques from identifying all CFTs, such as, path explosion, the jump targets affected by environment and input, etc., and identify the pros and cons of existing techniques, such as, linear dissemblers find more CFTs than recursive disassembler; pattern-based tools should incorporate rich code patterns; semantic interpretation of EVM operations is required to locate the target computed by arithmetic/bitwise operations, to name a few.

Moreover, we find that using execution traces can significantly improve these tools in terms of CFT identification. For example, the traces can complement the CFTs identified by OYENTE (a tool for discovering security vulnerabilities in smart contracts [5]) in about 40% of contracts. Leveraging this insight, we use the recovered execution traces from Ethereum to enhance OYENTE. The extensive experimental results show that our approach can reduce up to 30% of its false negatives. It is worth noting that the feasibility of recovering all execution traces of the deployed smart contracts is a unique feature in Ethereum. In contrast, although it is easy to collect some traces of traditional executables to evaluate binary analysis tools, it is extremely difficult, if not impossible, to get all execution traces of all binary executables. The insights obtained in our study highlight the ineffectiveness of static analysis techniques in identifying all CFT, and shed light on smart contract analysis. In summary, our work has three major contributions.

(1) To the best of our knowledge, it is the *first* large-scale analysis of CFT identification on smart contracts. We conduct a comprehensive examination on six widely-used tools with *all* deployed smart contracts and obtain many insights, some of which can be applied to other tools.

(2) We carefully instrument EVM to recover *all* execution traces of smart contracts and use them to evaluate the tools, and find that they can significantly complement the CFTs discovered by the tools.

(3) Motivated by the above observation, we enhance OYENTE with the CFTs extracted from execution traces. Extensive experiments show that the traces can largely reduce its false negatives.

Paper organization. Section II introduces background knowledge. Section III details the in-depth analysis of six tools in terms of their capabilities to identify CFTs. Section IV describes how we recover all execution traces and the comparison result of CFTs identified by tools and those from the traces. Section V presents how OYENTE can be enhanced with the traces. We discuss some possible limitations of our study in Section VI. After introducing related studies in Section VII, we conclude the paper in Section VIII.

II. Background

A. Ethereum

Ethereum blockchain has millions of blocks which are linked together. A block can contain many transactions, each of which is a message carrying important information (e.g. the bytecode of a smart contract) sent to the blockchain. To execute a smart contract, the caller should send a transaction to trigger the execution [24]. An Ethereum client is responsible for validating transactions and mining (i.e., producing blocks), and therefore a client will download all historical transactions from the blockchain and executes all transactions in order [24].

B. Smart Contract

Smart contract is an autonomous program [1], which is often developed in a high-level language (e.g., Solidity [2]) and then compiled into the bytecode that can be executed by EVM. Being a stack-based virtual machine, EVM maintains a

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on August 12,2020 at 13:43:26 UTC from IEEE Xplore. Restrictions apply.

stack to store the parameters and results of EVM operations. The size of a stack item ranges from 1 byte to 32 bytes [1], depending on the EVM operation that pushes the item onto the stack. The *memory* in EVM is an intrinsic data structure for storing the data attached in a transaction and the return value of the invoked contract [1]. A contract also has a database-like space, called *storage*, for storing persistent information (e.g., global variables, bytecode of smart contracts) [1].

A smart contract should be deployed to the blockchain before others can use it. In particular, the developer sends the bytecode of the contract (rather than the source code) to the blockchain. Although some developers submit the source code of their smart contracts to any blockchain explorers (e.g., Etherscan [25]) for code validation, the proportion of open-source smart contracts is less than 1% of all deployed contracts [26]. Once a contract is deployed to the blockchain, it will be associated with its address. To invoke a contract, its address should be specified.

C. CFT: Control Flow Transfer

The bytecode of a smart contract consists of multiple EVM operations. EVM supports 130+ EVM operations and eight out of them can result in CFTs. Two jump operations (i.e., JUMP, JUMPI) can result in intra-contract CFTs, which jump from an EVM operation to another EVM operation in the same contract [1] whereas the other six operations (i.e., CREATE, CALL, CALLCODE, DELEGATECALL, STATICCALL, SELFDESTRUCT) lead to inter-contract CFTs, which transfer from an EVM operation of a contract to the EVM operation of another contract [1]. This work concentrates on intra-contract CFTs, and we leave the inter-contract CFTs to future work. Note that it would not be a serious limitation of our study because the interactions of contracts are infrequent [27].

EVM stores the jump target on the stack. JUMP is an unconditional jump whose jump target is specified by the stack's top item [1]. JUMPI is a conditional jump that transfers the control flow to the target specified by the top stack item if the second stack item is not zero; otherwise the operation following the JUMPI will be executed next [1]. A jump target is marked by JUMPDEST [1]. Note that an intra-contract function invocation (e.g., a function calls another function in the same contract) is compiled into JUMP, and CALL is only for inter-contract invocation (e.g., a contract calls another contract). We represent a CFT in a smart contract as a tuple $\langle pc_s, pc_t \rangle$, where pc_s is the program counter of a JUMP or a JUMPI, and pc_t is the program counter of the corresponding JUMPDEST.

We summarize unique features of smart contracts compared to x86 binaries, which make it difficult to identify CFTs of smart contracts. First, smart contracts are compiled into EVM bytecode, which is a new instruction set without extensive studies. Second, both branch statements (e.g., 'if...else') and intra-contract function invocations are compiled into jump operations, whereas x86 uses CALL to invoke a function which distinguishes a function invocation from a jump. Third, the jump target is stored on the stack rather than encoded in the jump operation and therefore we cannot correlate a jump operation with its jump target by parsing the jump operation. In x86, the jump/call target of a direct jump/call is encoded in the jump/call instruction. Since there are various ways to push the jump target on the stack (Section II-D), correlating a jump operation with its jump target is non-trivial.

D. Motivating Examples

Fig. 1 presents six bytecode snippets extracted from deployed smart contracts to illustrate that *CFT identification of smart contracts can be quite challenging*. In Fig. 1(a), whether the contract jumps after executing JUMPI (Line 3) depends on the result of EQ (Line 1). EQ pushes 1 on the stack top if the top two stack items are equal; otherwise, it pushes 0 [1]. The jump target is the result of PUSH2 that pushes two bytes (i.e., 0x009d, the jump target) onto the stack. A tool can identify such CFTs by looking for the pattern PUSH2/JUMPI in contracts.

```
1  EQ
2  PUSH2 0x009d
3  JUMPI
      (a)
```

```
1  PUSH2 0x018e
...
2  POP
3  SWAP1
4  JUMP
      (b)
```

```
1  PUSH4 0xffffffff
2  PUSH2 0x0536
3  AND
4  JUMP
      (c)
```

```
1  ISZERO
2  PC
3  JUMPI
      (d)
```

```
1  PUSH1 0x40
2  MLOAD
3  JUMP
      (e)
```

```
1  PUSH1 0x04
2  SLOAD
3  PUSH2 0x1356
4  SWAP1
5  PUSH10 0x01...
6  SWAP1
7  DIV
8  PUSH4 0xffffffff
9  AND
10 JUMP
      (f)
```

Fig. 1. Six examples of CFTs

Fig. 1(b) illustrates that the push operation (Line 1) that pushes the jump target onto the stack can be located far away from the jump operation (Line 4). To identify such CFT, a tool needs to simulate stack operations or apply some static analysis techniques (e.g., reaching definition analysis [28]). Fig. 1(c) shows that the jump target is not directly encoded in the bytecode; instead, it is the outcome of a bitwise operation (Line 3). Therefore, to obtain the jump target, a tool should execute EVM operations or at least interpret the semantics of EVM operations. Fig. 1(d) sets the jump target of JUMPI (Line 2) by executing PC, which gets the current program counter (i.e., Line 2) and pushes the value onto the stack [1]. To handle such case, a tool needs to interpret the semantics of PC and locate the operation to get the jump target, because the target is not encoded in the bytecode.

In Fig. 1(e), the control flow is transferred by executing JUMP (Line 2) whose target is read from the memory (Line 2). MLOAD reads a value (0x40, Line 1) whose location is given in the top stack item from the memory and pushes the value onto the stack [1]. Fig. 1(f) presents a more complicated example, which reads a value from the storage by executing SLOAD (Line 2) and computes the jump target based on the value after a couple of EVM operations (Lines 3 to 9). Please recall that the memory stores the data from a transaction, and the storage can be used for storing contract runtime information (e.g., global variables) (Section II-B). Hence, it is difficult to determine such jump targets without executing the contract in practice. Even the simulation of EVM operations is unlikely to tackle the last two cases, because the jump targets are determined by the runtime values stored in the memory or the storage.

III. In-depth Analysis of CFT Identification

This section first describes our method to collect all smart contracts that have been deployed on the blockchain (Section

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on August 12,2020 at 13:43:26 UTC from IEEE Xplore. Restrictions apply.

III-A), followed by introducing the existing techniques for CFT identification (Section III-B). After presenting the selection of tested tools (Section III-C), we show the experimental results (Section III-D). In the end, we discuss the capabilities of different techniques for identifying CFTs of smart contracts (Section III-E).

A. Smart Contract Collection

A simple method to collect the bytecode of a deployed smart contract is to call the API, web3.eth.getCode() provided by an Ethereum client, given the address of a contract [29]. However, the API takes in the address of the queried contract [29] but unfortunately, it is not easy to obtain the addresses of all contracts. Moreover, the API can get the bytecode of a smart contract that have been removed providing a proper block number [29], however, for each removed contract, there is no easy way to know at which block (i.e., when) it is removed.

We collect the bytecode of all contracts by instrumenting an Ethereum client. After inspecting the source code of Geth (the most popular Ethereum client), we learn that the function evm.Create() is invoked for deploying contract and its return value is the contract address. Therefore, we obtain the address of a created contract before evm.Create() returns. evm.Create() calls evm.StateDB.SetCode() to store the contract bytecode (i.e., first parameter of evm.StateDB.SetCode()) in the storage. Therefore, to record contract bytecode, we add code in evm.Create() to log the first parameter of evm.StateDB.SetCode(). Our approach can even collect the bytecode of the contracts that have been deployed and then removed from the blockchain, because it captures all historical contract deployment activities. Consequently, we collect 4,979,625 smart contracts since the launching of Ethereum (Jul. 30th, 2015) to Feb. 10th, 2018.

B. Existing Techniques for CFT Identification

Before introducing the smart contract tools selected for this empirical study, we first summarize the techniques adopted by existing tools to identify CFTs of smart contracts as follows.

Symbolic Execution. SE uses symbolic values, instead of concrete values, as input and then represents the values of program variables as symbolic expressions over the symbolic input values [30]. SE is often used for path exploration, leveraging a theorem prover to determine path feasibility. SE-based tools for smart contracts usually simulate a stack and symbolically execute EVM operations, thus the jump target could be obtained from the simulated stack.

Pattern Recognition. EVM provides 32 push operations, PUSH*x* ($1 \leq x \leq 32$) which pushes an *x*-byte value on the stack [1]. Since the jump target is stored on the stack, a PUSH*x* right before a jump operation will push the jump target onto the stack. Based on our observation, a pattern recognition technique discovers CFTs by first looking for such code patterns (e.g., PUSH*x*/JUMP and PUSH*x*/JUMPI) in a contract and then extracting the jump target from PUSH*x*.

Light-weight static analysis. Some light-weight static analysis techniques (e.g., reaching definition analysis [8] and def-use analysis [15]) have been applied in CFT identification. For example, when reaching definition analysis is used, each EVM operation is annotated with a set of variables that are visible to this operation, and for every such variable, the position in code where the variable was last assigned before the operation is recorded [26]. It annotates every operation with the stack layout before the execution of the operation [26]. This stack layout contains references rather than actual values to the operations that have produced this stack item [26]. Based on light-weight static analysis, the push operation for setting the jump target can be identified, even if it is not located immediately before the jump operation.

C. Selected Tools for Smart Contract Analysis

Although a number of studies [5]–[16] present various tools that can identify CFTs in smart contracts, we filter out the tools that are neither publicly available nor capable of handling the bytecode of smart contracts. Eventually, we select six popular open-source tools [5]–[10] which can process EVM bytecode directly. OYENTE [5], MAIAN [7] and Mythril [6] discover vulnerabilities of smart contracts; evmdis [8], Miasm [9], and Porosity [10] are three reverse engineering tools for smart contracts.We classify these tools into two categories: path-sensitive tools including OYENTE [5], MAIAN [7] and Mythril [6] which apply SE to determine path feasibility, and path-insensitive tools including evmdis [8], Miasm [9], and Porosity [10]. Since path-insensitive tools will not check the feasibility of a program path, they may find CFTs that cannot be executed in practice. In contrast, path-sensitive tools only consider the CFTs that are discovered during path exploration. evmdis [8] is a disassembler which leverages reaching definition analysis. Miasm [9] is also a disassembler. Porosity [10] is a decompiler for EVM bytecode. The latter two apply pattern recognition to identify CFTs.

When running those tools to process contract bytecode, we find that OYENTE and MAIAN crash in processing 27,131 (0.5%) and 115,286 (2.3%) contracts, respectively. The other tools do not crash because they catch all exceptions. We find three causes (i.e., malformed bytecode, unsupported operations, and solver exceptions) of crashes and fix them because we are more interested in technical capabilities than the imperfection of implementations. Note that we do not resolve other (potential) bugs that do not lead to crashes.

D. Experimental Results of Selected Tools

This section presents the following experimental results: the number of CFTs identified by each tool, the number of contracts that different tools output the same result, and the number of CFTs that can be identified by all tools.

We use cft to represent a control flow transfer in a contract. Let $\mathcal{C}$ be the set of smart contracts analyzed in this study and $||$ denote the number of items in a set. Hence, $|\mathcal{C}| = 4,979,625$. We use $CFT_{c_i}^{OY}$, $CFT_{c_i}^{MA}$, $CFT_{c_i}^{MY}$, $CFT_{c_i}^{EV}$, $CFT_{c_i}^{MI}$ and $CFT_{c_i}^{PO}$ to stand for the CFTs in the contract c_i which are identified by OYENTE, MAIAN, Mythril, evmdis, Miasm and Porosity, respectively.

Number of CFTs. Fig. 2 shows the number of CFTs obtained by six tools. The average numbers are marked in this figure. We have several observations after inspecting the results and the source code of these tools. First, MAIAN finds the fewest CFTs because it stops path exploration once it finds a vulnerability. Hence, MAIAN cannot identify all CFTs since some

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on August 12,2020 at 13:43:26 UTC from IEEE Xplore. Restrictions apply.

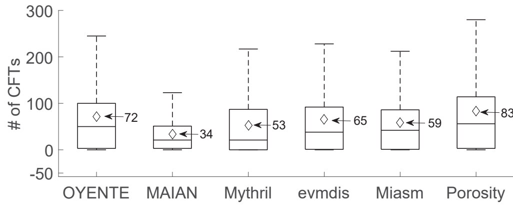

Fig. 2. Number of CFTs identified by the studied tools

paths are not explored. Besides, since MAIAN does not handle the contracts without either CALL, CALLCODE, DELEGATECALL, or SELFDESTRUCT, it finds zero CFT for such contracts. MAIAN adopts this strategy because it aims to detect three kinds of vulnerabilities, which include the aforementioned four EVM operations, rather than exploring all program paths.

Second, OYENTE finds more CFTs than Mythril due to the difference in their configurations although they both use SE. More precisely, since SE-based tools suffer from path explosion, a common mitigation is to set a maximum path depth, which refers to the max number of jump operations that a path can execute. By doing so, an SE-based tool terminates a path if its depth reaches the threshold. OYENTE can explore more paths than Mythril, because the former sets the maximum depth to 50 but the latter sets it to 12. Note that although MAIAN sets the maximum depth to 60, it discovers fewest CFTs since it does not handle some contracts and stops analysis once a vulnerability is found.

Third, Miasm and evmdis do not find as many CFTs as Porosity does, although they are all path-insensitive tools. We find that their differences lie in processing contract bytecode. More precisely, Miasm and evmdis are *recursive* disassemblers, which process contract bytecode following its control flow. More precisely, they disassemble the contract bytecode from the first operation, and then try to find the jump target when encountering a jump operation. If found, they continue to analyze from the EVM operation at the jump target. Otherwise, they stop analysis if they cannot determine a jump target. Contrarily, Porosity implements a *linear* disassembler, which does not follow the control flow of contract bytecode; instead, it checks *each* jump operation in the contract bytecode. Therefore, Porosity may find some CFTs whose jump operations are failed to be reached by Miasm. We then manually check the CFTs that are identified by Porosity but not discovered by OYENTE, and find that Porosity includes infeasible CFTs that will not be executed. OYENTE does not include such CFTs, because it checks path feasibility.

Finding 1: The six analyzed tools identify different numbers of CFTs due to their diverse techniques or configurations. Linear disassemblers find more CFTs than recursive disassemblers since the latter stops if a jump target cannot be determined.

$AVE(|CFT_{c_i}^{OY}|)$, $AVE(|CFT_{c_i}^{MA}|)$, $AVE(|CFT_{c_i}^{MY}|)$ are used for representing the average numbers of CFTs for all smart contracts identified by three path-sensitive tools. Fig. 3 shows the average number of CFTs identified by path-sensitive tools. The *x* axis is the number of jump operations, which suggests the complexity of a smart contract and is calculated by scanning contract bytecode. The *y* axis is the average number. Thus, a point (*x*, *y*) in this figure means that the average number of CFTs identified by a tool is *y* if the contracts contain *x* jump operations. Fig. 3 shows that the number of CFTs identified by three tools increases almost linearly when the number of jump operations increases from 1 to about 200. It is reasonable because a more complicated smart contract contains more CFTs. However, the linear trend does not persist when the number of jump operations is larger than 200. We find that they set four kinds of termination conditions to mitigate path explosion. That is, they will stop analysis if (1) it meets its objective; (2) it explores all paths whose depths are lower than the maximum path depth; (3) analysis time is expired; or (4) the number of BBs reaches a preset threshold. For example, MAIAN sets the maximum path depth as 60, and stops analyzing a contract if SE has executed 2,000 BBs (a BB will be counted multiple times if it is executed repeatedly). Besides setting the path depth as 12, Mythril gives 60s to each contract. OYENTE sets the timeout of each contract as 50s, and it terminates exploring a program path if a BB is executed more than 10 times.

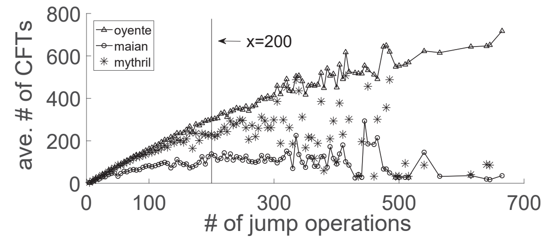

Fig. 3. Number of CFTs identified by path-sensitive tools

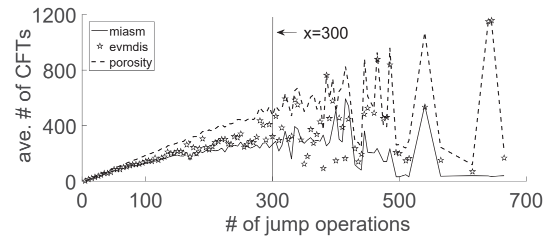

Fig. 4. Number of CFTs identified by path-insensitive tools

$AVE(|CFT_{c_i}^{EV}|)$, $AVE(|CFT_{c_i}^{MI}|)$, $AVE(|CFT_{c_i}^{PO}|)$ denote the average numbers of CFTs for all smart contracts identified by three path-insensitive tools. Fig. 4 presents the average numbers with the same *x* axis and *y* axis as Fig. 3. We can see that the average number discovered by three tools increases when the number of jump operations increases from 1 to about 300. However, none of them keeps the increasing trend, when the number of jump operations is larger than 300. We find several reasons. Miasm and evmdis stop analysis when they encounter a jump operation whose jump target cannot be determined by them. Hence, much code may remain unanalyzed when the two tools process complicated smart contracts. Besides, we observe that the code patterns recognized by Porosity are incomplete. More precisely, it only handles PUSH1 and PUSH2 to extract the jump target. Note that EVM supports 32 push operations [1] and a complicated contract is more likely to contain CFTs whose jump targets need more than two bytes. Therefore, Porosity may miss many CFTs in complicated contracts with many jump operations.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on August 12,2020 at 13:43:26 UTC from IEEE Xplore. Restrictions apply.

Finding 2: The six analyzed tools do not perform well in analyzing complicated contracts because of path explosion, premature stop, or incomplete code patterns.

Fig. 5 presents the size (in byte) of contract bytecode and the number of jump operations containing in a contract, which are averaged on the contracts created every week from the launching of Ethereum. It shows that the size and the number has an increasing trend over time. This observation highlights the need for the ability to analyze complicated smart contracts.

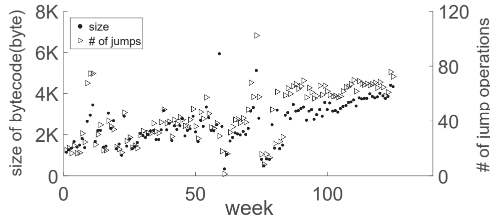

Fig. 5. Size of contract bytecode and number of jump operations in a contract

Contracts that different tools output the same result. If $CFT_{c_i}^{OY} = CFT_{c_i}^{MA}$, all CFTs in c_i identified by `OYENTE` are also discovered by `MAIAN`, and vice versa. Let $\mathcal{C}^{sym} = \{c_i | c_i \in \mathcal{C}, CFT_{c_i}^{OY} = CFT_{c_i}^{MA} = CFT_{c_i}^{MY}\}$. For each contract in $\mathcal{C}^{sym}$, `OYENTE`, `MAIAN` and `Mythril` have the same result in identifying CFTs. Let $\mathcal{C}^{oth} = \{c_i | c_i \in \mathcal{C}, CFT_{c_i}^{EV} = CFT_{c_i}^{MI} = CFT_{c_i}^{PO}\}$ and $\mathcal{C}^{all} = \{c_i | c_i \in \mathcal{C}, CFT_{c_i}^{OY} = CFT_{c_i}^{MA} = CFT_{c_i}^{MY} = CFT_{c_i}^{EV} = CFT_{c_i}^{MI} = CFT_{c_i}^{PO}\}$. They are the sets including the contracts that the three path-insensitive tools and all tools produce the same result, respectively. We obtain the following results: $|\mathcal{C}^{sym}| = 359,583$, $|\mathcal{C}^{sym}|/|\mathcal{C}| = 7\%$, $|\mathcal{C}^{oth}|/|\mathcal{C}| = 5\%$, and $|\mathcal{C}^{all}|/|\mathcal{C}| = 1.7\%$. That is to say, for a very small proportion of smart contracts, these tools find the same CFTs.

We then analyze path-sensitive tools and path-insensitive tools, separately. Fig. 6 presents the proportion of contracts that any two tools find the same CFTs in each contract. We find that `evmdis` produces the same results as `Porosity` for about 46% of smart contracts, which is the highest ratio compared to other combinations. The reason is that the CFTs recognized by code patterns can also be identified by reaching definition analysis, because the latter can find all push operations that push jump targets onto the stack. Moreover, any other two tools produce quite different results. The reason is that all tools except `Miasm` checks whether the operation at the jump target is a JUMPDEST. Therefore, all tools except `Miasm` exclude the CFTs with invalid jump targets (i.e., does not jump to a JUMPDEST). The three SE-based tools result in the same result for only 7% of contracts (not shown in this figure), since `MAIAN` stops if it discovers a vulnerability, and `OYENTE`, `Mythril` set different path depths.

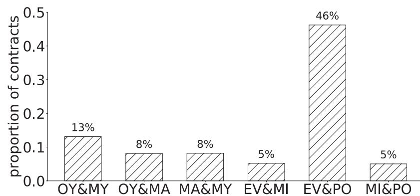

Fig. 6. Proportion of contracts that two tools find the same CFTs

Finding 3: The six analyzed tools identify the same CFTs for only 1.7% of contracts due to different techniques and configurations.

CFTs that can be identified by all tools. We further investigate the set $\mathcal{C}^{sym}$. The *x* axis in Fig. 7 is the number of jump operations, and the *y* axis represents the proportion of contracts. Therefore, a point (*x*, *y*) in this figure indicates that there are $y \times |\mathcal{C}^{sym}|$ contracts belonging to $\mathcal{C}^{sym}$, and each of them has *x* jump operations. The trend is that the proportion becomes smaller if the complexity of smart contracts increases. The outlier (1, 0.021) is because many contracts contain just one jump operation and do not contain the EVM operations concerned by `MAIAN`. Hence, `MAIAN` does not handle those contracts, resulting in a low ratio. The three path-insensitive tools have a similar trend and thus we omit the detail.

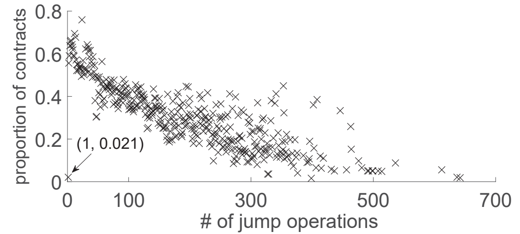

Fig. 7. Proportion of contracts that path-sensitive tools find the same CFTs

Let $\bigcap CFT_{c_i}^{oth} = CFT_{c_i}^{EV} \bigcap CFT_{c_i}^{MI} \bigcap CFT_{c_i}^{PO}$ represent the CFTs of the contract c_i that can be identified by all three path-insensitive tools. Besides, the set $\bigcup CFT_{c_i}^{oth} = CFT_{c_i}^{EV} \bigcup CFT_{c_i}^{MI} \bigcup CFT_{c_i}^{PO}$ indicates the CFTs of the contract c_i that can be identified by either of the three path-insensitive tools. For example, after analyzing a smart contract c_i, $CFT_{c_i}^{EV} = \{\langle 1,3 \rangle \langle 2,5 \rangle\}$, $CFT_{c_i}^{MI} = \{\langle 1,3 \rangle\}$ and $CFT_{c_i}^{PO} = \{\langle 1,3 \rangle \langle 2,7 \rangle\}$, and therefore $\bigcap CFT_{c_i}^{oth} = \{\langle 1,3 \rangle\}$, $\bigcup CFT_{c_i}^{oth} = \{\langle 1,3 \rangle \langle 2,5 \rangle, \langle 2,7 \rangle\}$.

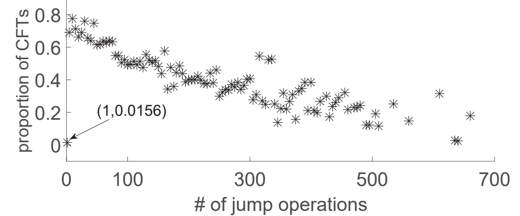

Fig. 8. Proportion of CFTs identified by all path-insensitive tools

We then investigate the set $\mathcal{C} - \mathcal{C}^{oth}$ using the above notations. $\mathcal{C} - \mathcal{C}^{oth}$ includes the contracts that for each contract, at least two path-insensitive tools do not find the same set of CFTs. The *x* axis in Fig. 8 is the number of jump operations, and the *y* axis is a proportion, which is computed by $AVE(|\bigcap CFT_{c_i}^{oth}|/|\bigcup CFT_{c_i}^{oth}|)$, indicating the average proportion of CFTs identified by all path-insensitive tools to the CFTs identified by either path-insensitive tools. Reconsider the example above, the proportion is 1/3. Therefore, a point (*x*, *y*) in this plot indicates that the CFTs of a contract identified by all three path-insensitive tools accounts for *y* of the CFTs identified by either path-insensitive tool, if it contains *x* jump operations. The observation is that the proportion becomes smaller if the complexity of smart contracts increases. The outlier (1, 0.0156) is due to the same reason with the

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on August 12,2020 at 13:43:26 UTC from IEEE Xplore. Restrictions apply.

outlier in Fig. 7. The three path-sensitive tools present a similar trend, thus we omit the detail due to page limit.

Finding 4: It becomes more difficult for tools to agree with each other when the complexity of contracts increases mainly because much code in complicated contracts remains unexplored when tools stop.

E. Discussion

Based on the experimental results, this section discusses the capabilities of different techniques listed in Section III-B for identifying CFTs of smart contracts.

Will the tools output infeasible CFT? An infeasible CFT cannot be taken during execution. The SE-based tools check path feasibility by invoking a theorem prover so that they will not output infeasible CFTs. However, the path-insensitive tools may include infeasible CFTs because they do not check path feasibility.

Disassembling mode. A Linear disassembler can identify more CFTs than a recursive disassembler, because the latter has to stop if a jump target cannot be determined while the former can still discover the remaining CFTs.

Can the tools handle complex contracts. All three SE-based tools are not scalable to complicated contracts due to path explosion. Experiments show that the three path-insensitive tools also do not perform well in processing complicated smart contracts (Fig. 4), because they choose the recursive disassembling mode or do not include sufficient patterns.

CFT types. Existing techniques perform differently to handle different types of CFTs.

(a) Push immediately before jump. All three techniques (i.e., SE, pattern recognition and light-weight static analysis) can identify the jump target that is pushed on the stack immediately before a jump operation, as shown in Fig. 1(a).

(b) Push elsewhere. The jump target can be pushed on the stack by a push operation located elsewhere rather than the location exactly before a jump operation, as shown in Fig. 1(b). SE can identify such jump target if it covers the corresponding jump operation, because SE simulates a stack and interprets EVM operations. Light-weight static analysis techniques (e.g., reaching definition analysis, def-use analysis) can also handle this case, since they can identify use-define relationship. That is, they can locate the push operation which pushes the jump target on the stack. However, pattern recognition fails because such case does not match the two code patterns, PUSH*x*/JUMP and PUSH*x*/JUMPI.

(c) Target by computation. If the jump target is computed by arithmetic/bitwise operations, as shown in Fig. 1(c), SE can handle this case because it executes EVM operations symbolically. Reaching definition analysis cannot discover such jump target since the target is not encoded in a push operation. The CFT cannot be determined by pattern recognition because such case does not match the two code patterns.

(d) Target needs interpretation. The jump target can be determined by interpreting the semantics of EVM operations, as shown in Fig. 1(d). For the same reasons as the target by computation, only SE can handle such case.

(e) Target affected by environment. In this case, the jump target is affected by the environment (e.g., memory, storage). All three techniques cannot identify such jump target because static analysis cannot know the runtime values in the memory and the storage.

IV. Trace-based CFT Identification

We extract CFTs from execution traces of smart contracts to examine whether execution traces can complement the tools.

A. Trace Collection

Execution trace. The execution trace logs the execution information of every executed EVM operation in a smart contract. The recorded information consists of the executed EVM operation, the program counter, the stack/memory/storage read/written by the operation, and the other data read from the blockchain (e.g., mining difficulty, block number).

Ethereum Client instrumentation. An approach to obtain traces is invoking a Web3 API, web3.debug.traceTransaction() which takes in the hash of a transaction and outputs the trace triggered by that transaction [31]. However, there is not easy to obtain all transaction hashes. More severely, the API runs slowly and after inspecting the source code of Ethereum, we find that before executing the queried transaction, this API has to initialize the runtime environment, construct the correct state before the execution of the block containing the queried transaction, and then replay the preceding transactions before the queried transaction in the same block. Besides, Web3 APIs use Remote Procedure Calls to communicate with an Ethereum node, which introduces further delay.

We propose to instrument an Ethereum client to recover execution traces based on the following observations. A client will execute all historical transactions and the execution of smart contracts is triggered by transactions [24]. After investigating the source code of Geth, we reveal that Geth provides a handler for each EVM operation. Therefore, we insert some code into all handlers to record execution information. Fig. 9 shows how to log a CFT when executing JUMPI. opJumpi() (Line 1) is the handler (simplified for presentation) for executing JUMPI. The jump target and the flag are popped from the stack (Line 2). We record the program counter of the JUMPI at Line 3. Then, the program counter is updated accordingly (Lines 5, 6). We record the new program counter at Line 7. Therefore, the CFT is logged, which jumps from the old program counter to the new one. Eventually, we collect 63,594,036 execution traces from the launching of Ethereum to Feb. 10th, 2018.

```
1  func opJumpi(pc *uint64, ..., stack *Stack)(){
2    pos, cond := stack.pop(), stack.pop()
3    log("current PC"+*pc)
4    if cond.Sign() != 0 {...
5      *pc = pos.Uint64()
6    } else {*pc++}
7    log("jump target"+*pc)}
```

Fig. 9. Instrumentation of JUMPI's handler

CFT covered by traces. Identifying the CFTs covered by traces needs to instrument the handlers for JUMP and JUMPI, because the jump target is the top stack item when executing JUMP or JUMPI.

Trace splitting. A contract can invoke another contract, and therefore an execution trace can contain CFTs from multiple contracts. Therefore, we need to split the execution trace into several sub-traces and each sub-trace corresponds to

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on August 12,2020 at 13:43:26 UTC from IEEE Xplore. Restrictions apply.

a contract. To do so, we need to find the beginning and ending of executing a contract. There are four EVM operations (i.e., CALL, CALLCODE, DELEGATECALL and STATICCALL) that a contract calls another contract [1]. Therefore, we instrument the handlers for those four operations.

```
1  func opCall(evm *EVM, contract *Contract, stack *Stack...)(){
2    stack.pop()
3    addr := stack.pop()
4    toAddr := common.BigToAddress(addr)
     ...
5    log("Contract start"+toAddr)
6    ret, returnGas, err := evm.Call(contract, toAddr, ...)
7    log("Contract end")...}
```

Fig. 10. Mark the beginning and ending of executing a contract

Fig. 10 shows the code snippet of opCall(), which is the handler for executing CALL. Since the other three handlers are similar with opCall(), we omit their details. The second item on the stack is the address of the invoked contract (Line 3), so we mark the beginning of executing *B* at Line 5. The contract *B* is executed using evm.Call() (Line 6), and hence we mark the ending of executing *B* at Line 7.

B. Evaluation Results

This section presents to what extent, execution traces can complement the CFTs identified by the studied tools, including the number of contracts whose CFTs can be complemented by traces, and the number of CFTs complemented by traces.

Contracts whose CFTs can be complemented by execution traces. We use $CFT_{c_i}^{TR}$ to represent the CFTs in contract c_i extracted from execution traces. We then use $\mathcal{C}_{OY}^{TR}$ to represent the contracts that some extracted CFTs are not discovered by OYENTE. Thus, $\mathcal{C}_{OY}^{TR} = \{c_i | \exists cft, cft \in CFT_{c_i}^{TR}, cft \notin CFT_{c_i}^{OY}\}$. We use the value $|\mathcal{C}_{OY}^{TR}|/|\mathcal{C}|$ to stand for the proportion of such contracts to the total analyzed contracts. We define $\mathcal{C}_{MA}^{TR}$, $\mathcal{C}_{MY}^{TR}$, $\mathcal{C}_{EV}^{TR}$, $\mathcal{C}_{MI}^{TR}$ and $\mathcal{C}_{PO}^{TR}$ accordingly, and omit their definitions due to page limit. The proportions according to the studied tools are shown in Fig. 11. This figure shows that for many smart contracts, execution traces can cover some CFTs that are not discovered by the six tools.

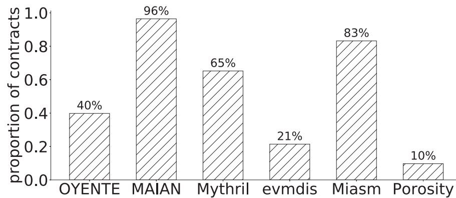

Fig. 11. Proportion of contracts that execution traces can complement their CFTs discovered by the studied tools

We further have several specific observations. First, execution traces complement Porosity in about 10% of contracts, even if it finds the most CFTs (Fig. 2). Second, the proportion for Miasm is fairly high, 83%, because it stops analysis if the immediate operation before a jump operation is not a push operation, thus it will leave much code unanalyzed. For evmdis, the proportion (21%) is much lower, compared to Miasm, even if they both stop analysis if they cannot determine a jump target. The reason is that evmdis applies reaching definition analysis, so it can find the CFT which cannot be identified by Miasm. An example of such CFT is shown in Fig. 1(b) that the jump target is pushed on the stack by a push operation far before the jump operation. Besides, for MAIAN, almost the CFTs of all (96%) contracts can be complemented by traces. The reason is that MAIAN does not process the contracts without some specific operations, and it stops analysis if a bug is found. For Mythril, the proportion is higher than OYENTE (65% vs. 40%). The reason should be that OYENTE sets a larger path depth than Mythril (50 vs. 12).

Number of CFTs complemented by execution traces. We use $CFT_{c_i}^{TR/OY}$ to represent the CFTs which are extracted from the execution traces, but not discovered by OYENTE. Thus, $CFT_{c_i}^{TR/OY} = \{cft | cft \in CFT_{c_i}^{TR}, cft \notin CFT_{c_i}^{OY}\}$. Hence, $|CFT_{c_i}^{TR/OY}|$ stands for the number of such CFTs in c_i. We define such numbers for the other five tools accordingly and we omit the definitions due to page limit. Fig. 12 presents such numbers of the studied tools, where we mark the averages and maximum numbers. Note that this figure just includes the contracts, some CFTs of which are executed by traces but not identified by tools. The average numbers range from 10 to 17, however, the maximum numbers are somewhat surprising, which range from 122 to 1,125. Therefore, the observation is that lots of CFTs can be complemented by execution traces, because these CFTs are failed to be identified by the six tools. We then investigate the maximum numbers in detail.

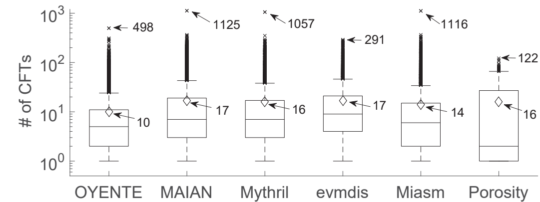

Fig. 12. Numbers of the CFTs covered by traces but not discovered by tools

The maximum numbers of four tools (i.e., OYENTE, MAIAN, Mythril and Miasm) are related to the same contract, 0xDA16. The address of a contract is 20 bytes, and we use the first two bytes to refer to a contract to save space. The contract 0xDA16 is open source [32]. By inspecting its source code, we find nested loops and there are 121 'else if{}' statements in the inner loop. Besides, the size of the contract bytecode is 24,556 bytes, and we find 642 jump operations in its bytecode. We capture eight transactions invoking the contract and thus we collect eight execution traces. Execution traces add 1,125 CFTs to the CFTs identified by MAIAN, since the contract does not contain the operations concerned by MAIAN. Therefore, MAIAN finds zero CFT in the contract. Execution traces add 498 and 1,057 CFTs to the CFTs discovered by OYENTE and Mythril, respectively due to the relatively small path depths set by the tools. We find that the shortest trace consists of 657,051 EVM operations, and 45,862 out of them are jump operations. Note that 45,862 is far deeper the path depths (i.e., 50 and 12) set by the two tools. Execution traces add 1,116 CFTs to the CFTs discovered by Miasm, because the contract has two consecutive operations, POP and JUMP that Miasm fails to determine the jump target. Consequently, Miasm stops and leaves much code unanalyzed.

The maximum numbers of the other two tools (i.e., evmdis and Porosity) are related to the same contract, 0xe414. The size of the contract bytecode is 22,456 bytes, and we find 273 jump operations in its bytecode. We capture 240 transactions and thus we collect 240 execution traces. The traces add 291

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on August 12,2020 at 13:43:26 UTC from IEEE Xplore. Restrictions apply.

CFTs to the CFTs found by `evmdis` because we find a jump operation whose jump target is computed by an ADD operation rather than encoded in a push operation. Execution traces add 122 CFTs to the CFTs discovered by `Porosity`, since we find many jump operations whose jump targets are pushed on the stack by executing PUSH3 that is not handled by `Porosity`.

Finding 5: Execution traces effectively complement the studied tools in identifying CFTs, since no tool can find all CFTs and execution traces cover many CFTs which cannot be discovered by the studied tools.

C. Possible Usages

Besides evaluating the effectiveness of tools to identify CFTs, trace-based CFT identification has at least two possible usages. First, before deploying a smart contract, we can use fuzzing to generate traces for complementing its CFG obtained by static analysis. Second, after deploying a smart contract, recovering the traces of all invocations in blockchain and analyzing them allow us to conduct more accurate investigations (e.g., discovering vulnerabilities/errors, locating faults.). An example of the second usage is shown in Section V.

V. Enhancing OYENTE through Traces

We propose to enhance OYENTE [5] by traces. Section V-A presents a motivating example that execution traces improve the performance of OYENTE. Then, Section V-B describes our approach to enhance the path exploration module of OYENTE through the traces. We conduct a large-scale study on to what extent the traces can improve OYENTE in Section V-C.

```
1  function buyWanCoin(address recipient){
   ...
2    if(now < startTime && now >= earlyReserveBeginTime){...}
3    else{...
4      buyNormal(recipient);}
   ...}

5  function buyNormal(address recipient)internal{
6  ...
7    buyCommon(recipient, toFund, toCollect);}

8  function buyCommon(address recipient,
    uint toFund, uint wanTokenCollect)internal{
   ...
9    wanport.transfer(toFund);
   ...}
```

Fig. 13. OYENTE misses the timestamp dependence bug at Line 2, but execution traces help in discovering it

A. A Motivating Example

Fig. 13 shows the code snippet of a token (i.e., the cryptocurrency implemented as a smart contract), which has been deployed to the blockchain since Oct. 2th, 2017. This contract has 4,657 bytes bytecode, 442 BBs and 143 jump operations, and we recover 19,072 execution traces from the transactions invoking this contract. Manual analysis of its source code reveals that it is a vulnerable timestamp-dependent contract [5], which uses the block timestamp to control the execution of some critical operations. More precisely, the comparison at Line 2 uses 'now' in Solidity to get the timestamp of the current block [2]. Hence, the timestamp determines whether or not the critical function buyNormal() (Line 4) will be executed, which invokes buyCommon() (Line 7) to send out money (Line 9). Consequently, a malicious miner can adjust the block's timestamp to affect the execution of this contract.

By leveraging the execution traces, we discover this vulnerable timestamp-dependent contract whereas OYENTE misses it due to path explosion. More precisely, the function buyCommon() remains unexplored after OYENTE stops analysis. In contrast, execution traces complement the results of OYENTE, because some traces cover the three functions and trigger money transfer at Line 9. For example, in the trace of a transaction (0x10fb referred to by the first two bytes of its hash), we observe that after the execution of the 70th CFT, the contract sends out money. Since 70 is much larger than the maximum path depth (50) of OYENTE, OYENTE cannot output correct results due to path explosion. In contrast, after running OYENTE on the traces, the vulnerability is discovered.

B. Enhancing Approach

```
Input: bc, CFT_TR
Output: report
1  CFT_OY = explore(bc)
2  for cft ∈ CFT_TR − CFT_OY
3    for trace ∈ traces(cft)
4      if trace ∉ replayed_TR
5        replay(bc,trace)
6        replayed_TR.append(trace)
7  return (report = Analysis())
```

Fig. 14. Enhancing OYENTE with the CFTs from traces.

Fig. 14 shows how we enhance OYENTE. It takes in the contract bytecode (*bc*) for analysis, the CFTs extracted from traces (CFT_{TR}) and feeds the analysis modules of OYENTE with new path conditions and the results of path exploration to generate a report on vulnerabilities. More precisely, we first record the CFTs (CFT_{OY}) discovered by the original OYENTE during path exploration (Line 1). For every CFT that is extracted from traces but not in CFT_{OY} (Line 2), we obtain the traces covering that CFT (Line 3). We replay them to construct path conditions, which are required to detect vulnerabilities. For instance, to detect timestamp-dependent contracts, OYENTE checks whether path conditions contain timestamps. Unlike path exploration, trace replaying does not need to solve path conditions, because a trace indicates a feasible path. The replayed trace will be recorded to prevent repeated replay (Line 6). We use the enhanced OYENTE to analyze all collected (4,979,625) contracts and present the results in the following subsection.

C. Results of Enhanced OYENTE

OYENTE discovers four kinds of vulnerabilities. Specifically, it detects *mishandled exceptions* by scanning contract bytecode. Besides, it detects the other three bugs by first exploring the paths of smart contracts and then checking whether these bugs exist in the explored paths. Therefore, the enhanced OYENTE can find the false negatives produced by the original OYENTE, if the latter misses some paths which contain bugs.

Transaction-Ordering Dependence (TOD). The executions of such contracts depend on the order of transactions, which can be manipulated by malicious miners [5]. The enhanced OYENTE discovers 521,330 TOD contracts whereas the original one only uncovers 80.4% of them (i.e., 460,626). That

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on August 12,2020 at 13:43:26 UTC from IEEE Xplore. Restrictions apply.

is, without enhancement, OYENTE has 60,704 false negatives ($11.6\% = 60,704/521,330$).

Timestamp Dependence. The enhanced OYENTE uncovers 183,668 such contracts whereas the original one only detect 84.3% of them (i.e., 154,871). Hence, the original OYENTE results in 28,797 false negatives ($15.7\% = 28,797/183,668$).

Mishandled Exceptions. Such contracts fail to conduct a proper check on the return value of calling another contract and thus may produce unexpected results because exceptions may be raised during the execution of the callee contract [5]. With/without enhancement, we detect the same amount (115,731) of such contracts. It is excepted because OYENTE scans the contract bytecode to locate the vulnerability pattern instead of exploring the paths to vulnerable codes and therefore the CFTs from execution traces have no effect in finding such vulnerability.

Reentrancy Vulnerability. A contract *A* has a reentrancy vulnerability if it calls another contract *B* and before the call returns, *B* calls *A* using the intermediate state of *A* [5]. A reentrancy vulnerability in TheDAO contract results in 60 million USD money loss [3]. The enhanced OYENTE finds 81,931 such vulnerable contracts whereas the original one only discovers 70% of them (i.e., 57,392). In other words, the original OYENTE leads to 24,539 false negatives ($30\% = 24,539/81,931$).

Summary. Execution traces can reduce the *false negative rate* of OYENTE up to 30%, because traces cover some program paths and CFTs that are not found by OYENTE.

VI. Discussion

This section discusses possible limitations of our study. First, the experimental results may be affected by the implementation flaws in the studied tools. To reduce such impact, we have fixed the bugs which result in crashes of these tools (Section III-C). Since our evaluation process has been automated through scripts, we could evaluate the new versions of these tools if any and compare them with existing ones to further mitigate such impact. Second, the quantitative experimental results are specific to the studied tools. It is worth noting that most insights obtained by our analysis are general to the techniques applied by those tools (Section III-E). To make our experimental results representative, we select the six widely-used open-source tools. Besides, the experimental results may also be influenced by the configurations (e.g., max path depth) of the studied tools. To reduce the influence, we evaluate those tools with their default configurations, assuming that the developers have tuned the parameters for best results. We will explore more configurations in future work. Moreover, we will release our data and program after paper publication for the ease of reproducing our results and evaluating other tools.

VII. Related Work

There are many program analysis tools [33], [34] for smart contracts (EVM bytecode), which identify CFTs during analysis. OYENTE [5], GASPER [11], MAIAN [7], Mythril [6], and teEther [16] apply SE to analyze smart contracts. evmdis finds jump targets using reaching definition analysis [8]. sCompile [15] depends on def-use analysis to discover CFTs. Miasm [9] and Porosity [10] discover CFTs by looking for two code patterns. Securify is a verifier for contract security, which constructs CFG during bytecode decompilation [13]. Fröwis and Böhme reuse the reaching definition analysis module in evmdis [8] to check whether the targets of inter-contract CFTs (e.g., by executing CALL) are affected by the environment [26]. ContractFuzzer discovers security bugs of EVM bytecode by fuzzing [17]. Osiris applies SE to reveal integer overflow vulnerabilities in EVM bytecode, and uses taint analysis to reduce false positives. Zeus is verifier which applies abstract interpretation and symbolic model checking to check security properties [14]. Based on SE, teEther discovers security bugs in EVM bytecode, and then generates transactions which can exploit the bugs [16].

Although lots of efforts [19]–[21] have been made to binary disassembly, it is still an unsolved problem since disassembly is generally undecidable [22]. A recent empirical study on nine state-of-the-art binary disassemblers shows that there are several technical challenges (e.g., inline data, jump table, alignment code, unreachable code, indirect calls) preventing disassembly to obtain accurate results [23]. Besides, it is quite different to disassemble EVM bytecode compared to binaries. Execution traces have been used in many domains of software engineering, e.g., bug diagnosis [35], feature location [36], test case extraction [37], task offloading [38], fuzzing [39], and software protection [40]. Our work differs in the goal of recording traces, which is used for evaluating and enhancing existing techniques in identifying CFTs, and in the way to prepare inputs for producing traces. To prepare inputs, the existing approaches either depend on human efforts [36], produce random inputs [38], or deploy on the devices of end users to collect user inputs [35]. Contrarily, we get the inputs from the blockchain because the blockchain stores all historical inputs (i.e., transactions) to the programs (i.e., smart contracts). Therefore, we obtain all historical traces of smart contracts that is extremely difficult to achieve this goal in other platforms (e.g., get all historical traces of a desktop software which has distributed to numerous users).

VIII. Conclusion

We conduct the first in-depth empirical study on the capabilities of existing techniques for identifying CFTs in smart contracts, and obtain many meaningful observations and insights. Exploiting the salient features of Ethereum, we collect all deployed smart contracts and recover all execution traces. By contrasting the CFTs discovered by existing tools and those extracted from the traces, we find that the traces can significantly complement the CFTs discovered by the tools. Leveraging this insight, we enhance OYENTE with execution traces, and the extensive experiments demonstrate that this approach can largely reduce its false negatives up to 30%. This study can benefit Ethereum users, developers and analysts. We will release our data and program after paper publication.

Acknowledgement

Ting Chen is partially supported by National Natural Science Foundation of China (61872057) and National Key R&D Program of China (2018YFB0804100). Ting Wang is partially supported by the National Science Foundation under Grant No.1566526 and 1718787.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on August 12,2020 at 13:43:26 UTC from IEEE Xplore. Restrictions apply.

References

[1] G. Wood. (2018) Ethereum: a secure decentralised generalised transaction ledger — byzantium version. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[2] Ethereum. (2018) Solidity documentation. [Online]. Available: http://solidity.readthedocs.io/en/v0.4.24/

[3] M. del Castillo. (2016) The dao attacked: Code issue leads to $60 million ether theft. [Online]. Available: https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft/

[4] A. Hertig. (2017) Ethereum client bug freezes user funds as fallout remains uncertain. [Online]. Available: https://www.coindesk.com/ethereum-client-bug-freezes-user-funds-fallout-remains-uncertain/

[5] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *CCS*, 2016.

[6] B. Mueller. (2018) Smashing ethereum smart contracts for fun and real profit. [Online]. Available: https://github.com/b-mueller/smashing-smart-contracts/blob/master/smashing-smart-contracts-1of1.pdf

[7] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. (2018) Finding the greedy, prodigal, and suicidal contracts at scale. [Online]. Available: https://arxiv.org/abs/1802.06038

[8] Arachnid. (2018) evmdis: an evm disassembler. [Online]. Available: https://github.com/Arachnid/evmdis

[9] J.-B. Cayrou. (2018) Reverse engineering framework in python. [Online]. Available: https://github.com/jbcayrou/miasm/tree/evm

[10] M. Suiche. (2017) Porosity: A decompiler for blockchain-based smart contracts bytecode. [Online]. Available: https://www.comae.io/reports/dc25-msuiche-Porosity-Decompiling-Ethereum-Smart-Contracts-wp.pdf

[11] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *SANER*, 2017.

[12] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *WETSEB*, 2018.

[13] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev. (2018) Securify: Practical security analysis of smart contracts. [Online]. Available: https://arxiv.org/abs/1806.01143

[14] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *NDSS*, 2018.

[15] J. Chang, B. Gao, H. Xiao, J. Sun, and Z. Yang. (2018) scompile: Critical path identification and analysis for smart contracts. [Online]. Available: https://arxiv.org/pdf/1808.00624.pdf

[16] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *USENIX Security*, 2018.

[17] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: fuzzing smart contracts for vulnerability detection," in *ASE*, 2018.

[18] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *ACSAC*, 2018.

[19] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *USENIX Security*, 2004.

[20] G. Tan and T. Jaeger, "Cfg construction soundness in control-flow integrity," in *PLAS*, 2017.

[21] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *S&P*, 2016.

[22] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," in *ECML-PKDD*, 2011.

[23] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *USENIX Security*, 2016.

[24] Ethereum. (2018) Ethereum homestead documentation. [Online]. Available: http://www.ethdocs.org/en/latest/

[25] Etherscan. (2018) Etherscan—the ethereum block explorer. [Online]. Available: https://etherscan.io/

[26] M. Fröwis and R. Böhme, "In code we trust?" in *DPM*, 2017.

[27] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhang, "Understanding ethereum via graph analysis," in *INFOCOM*, 2018.

[28] J.-F. Collard, *Reasoning about program transformations: imperative programming and flow of data*. Springer Science & Business Media, 2007.

[29] Ethereum. (2019) Javascript api. [Online]. Available: https://github.com/ethereum/wiki/wiki/JavaScript-API

[30] J. C. King, "Symbolic execution and program testing," vol. 19, pp. 385–394, 1976.

[31] Ethereum. (2017) Management apis. [Online]. Available: https://github.com/ethereum/go-ethereum/wiki/Management-APIs

[32] Etherscan. (2018) Contract, 0xda16251b2977f86cb8d4c3318e9c6f92d7fc1a8f. [Online]. Available: https://etherscan.io/address/0xDA16251B29\\77F86cB8d4C3318e9c6F92D7fC1A8f\#code

[33] R. Fontein, "Comparison of static analysis tooling for smart contracts on the evm," in *TSConIT*, 2018.

[34] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and tools for the static analysis of ethereum smart contracts," in *CAV*, 2018.

[35] W. Jin and A. Orso, "Bugredux: reproducing field failures for in-house debugging," in *ICSE*, 2012.

[36] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *ASE*, 2007.

[37] F. Křikava and J. Vitek, "Tests from traces: Automated unit test extraction for r," in *ISSTA*, 2018.

[38] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *EuroSys*, 2011.

[39] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for os kernels," in *USENIX Security*, 2017.

[40] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," in *ASPLOS*, 2017.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on August 12,2020 at 13:43:26 UTC from IEEE Xplore. Restrictions apply.