Hunting Vulnerable Smart Contracts via Graph Embedding Based Bytecode Matching

Jianjun Huang, Songming Han, Wei You, Wenchang Shi, Bin Liang, Jingzheng Wu and Yanjun Wu

***Abstract*—Smart contract vulnerabilities have attracted lots of concerns due to the resultant financial losses. Matching-based detection methods extrapolating known vulnerabilities to unknown have proven to be effective in other platforms. However, directly adopting the technique to smart contracts is obstructed by two issues, i.e., diversity of bytecode generation resulting from the rapid evolution of compilers and interference of noise code easily caused by the homogeneous business logics. To address the problems, we propose contract bytecode-oriented normalization and slicing techniques to augment bytecode matching. Specifically, we conduct data- and instruction-level normalizations to uniform the bytecode generated by different compilers, and enforce contract-specific slicing by tracking data- and control-flows with simulated bytecode executions to prune the noise code as far as possible. Based on the above techniques, we design an unsupervised graph embedding algorithm to encode the code graphs into quantitatively comparable vectors. The potentially vulnerable smart contracts can be identified by measuring the similarities between their vectors and known vulnerable ones. Our evaluations have shown the efficiency (0.47 seconds per contract on average), effectiveness (160 verified true positives) and high precision (91.95% for top-ranked). It is worth noting that, we also identify dozens of honeypot contracts, further demonstrating the capability of our method.**

***Index Terms*—Vulnerable Smart Contracts, Graph Embedding, Bytecode Matching**

I. Introduction

ALONG with the proliferation of Ethereum, smart contracts, programs running on Ethereum and mainly developed in Solidity, have attracted lots of security concerns. In fact, smart contract vulnerabilities have caused enormous financial damages. For example, vulnerabilities in theDAO [1] and BeautyChain [2] led to more than one billion worth of loss. Therefore, vulnerability detection for smart contracts becomes a critical task in recent years.

Matching-based methods have proven to be effective for detecting vulnerabilities, for example, in desktop and IoT applications [3]–[7]. The intuition behind such methods is to search similar implementations to known vulnerabilities. Unfortunately, adopting the matching-based technique to smart contracts is a nontrivial task. Two problems need to be addressed.

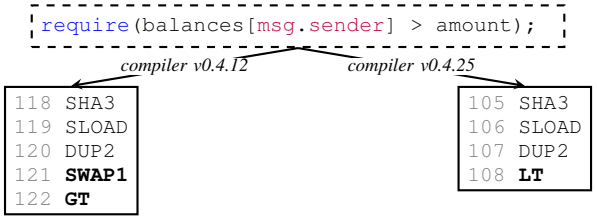


Fig. 1. Diverse bytecode resulting from different versions of the compiler.

First, studies have shown that only 1% of smart contracts are open source [8], so a practicable matching-based detection method should work with the bytecode. However, the rapid evolution of Solidity compilers has led to dozens of compiler versions and different versions may generate different bytecode even for the same source code pieces. The diversity of bytecode generation can impede bytecode matching. For instance, a vulnerable implementation can be missed just because it was compiled with a different version of the compiler and thus contains a large number of different instructions. We demonstrate the impact with the bytecode generated by two versions of the compiler for the statement in Figure 1. The older version emits 36 instructions for the statement while the newer produces 33, among which 28 instructions are the same. We show in Figure 1 two snippets of discrepancies, with different program counters and `SWAP1/GT` *v.s.* `LT` for the same comparison operation. Using the bytecode generated by the newer compiler as the query seed and the one by the older version as the target, the similarity is only 77.8% for the same source code.

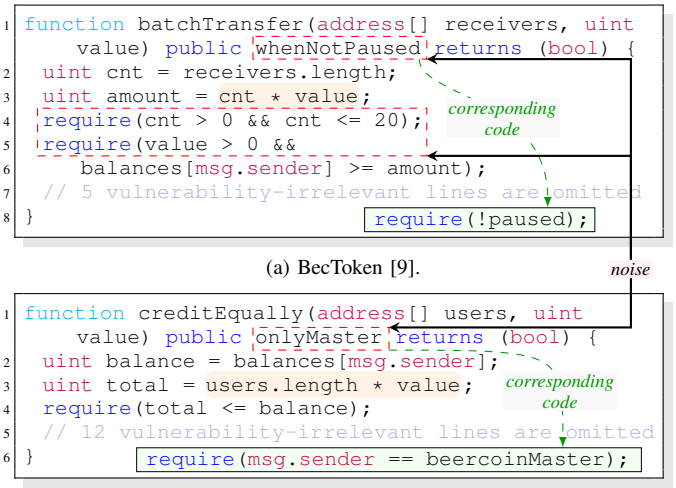
Second, a vulnerability generally involves very few statements but vulnerable smart contracts, either the seeds or targets, usually contain many statements that are irrelevant to the vulnerabilities. Those vulnerability-irrelevant statements, called *noise code*, can confuse code matching and thus mitigating their interference is an urgent task to be solved. Take Figure 2 as an example, in which the two functions suffer from the same kind of multiplication overflows (highlighted in orange). In each function, only three lines that perform calculation and insufficient protection are related to the corresponding vulnerability. All the others are irrelevant, including the boxed parts and the omitted sections, neither contributing to the overflow logic nor trying to ensure the multiplication safety. The presence of such noise code can prevent us effectively matching

Manuscript received XXXX xx, 2020; revised XXXX xx, 2020; accepted XXXX xx, 2020. Date of publication XXXX xx, 2020; date of current version XXXX xx, 202X. This work was supported in part by National Natural Science Foundation of China (NSFC) under grants U1836209 and 61802413, the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University of China under grants 19XNLG02 and 20XNLG03. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Ghassan Karame. *(Corresponding author: Bin Liang.)*

Jianjun Huang, Songming Han, Wei You, Wenchang Shi and Bin Liang are with the School of Information, Renmin University of China, Beijing 100872, China; and also with Key Laboratory of DEKE (Renmin University of China), MOE, China. (e-mail: hjj@ruc.edu.cn; 2013202516@ruc.edu.cn; youwei@ruc.edu.cn; wenchang@ruc.edu.cn; liangb@ruc.edu.cn).

Jingzheng Wu and Yanjun Wu are with the Institute of Software, Chinese Academy of Sciences, Beijing 100190, China. (e-mail: jingzheng08@iscas.ac.cn; yanjun@iscas.ac.cn).

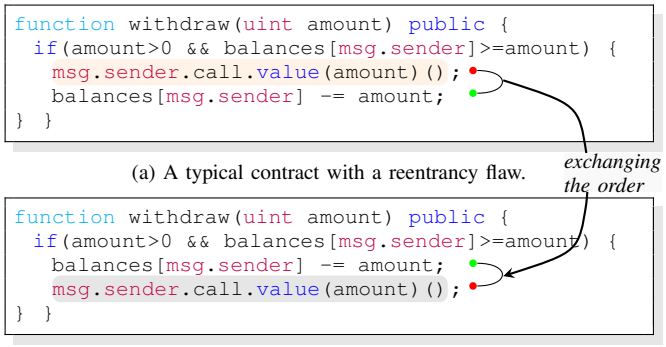
Digital Object Identifier 10.1109/TIFS.XXXX



(a) BecToken [9].

(b) Beercoin [10].

Fig. 2. Two contracts with the same multiplication overflow vulnerability.



(a) A typical contract with a reentrancy flaw.

(b) Simple fix of the above vulnerable function.

Fig. 3. Two functions with the same statements in different orders.

the known vulnerable BecToken with the one in Figure 2b, leading to missing a potential vulnerability. In fact, the bytecode matching similarity of only 51.7% cannot confidently report Beercoin as vulnerable. Apart from this situation, we must notice that, in practice, many contracts enforce homogeneous business logics due to limited application scenarios. It inevitably brings about a lot of similar implementations, such as ether depositing/withdrawal and token transfers. More specifically, balance checks (e.g., line 4 in Figure 2b) and operations (e.g., addition and subtraction), ether sending via *call* and some other common actions will occur frequently and are usually implemented in the same manner. Consequently, matching-based analysis of smart contracts is more susceptible to noise code than traditional applications. When the homogeneous noise code dominates the functions, the crucial operations directly related to the vulnerabilities will not receive sufficient attention in code matching. In other words, vulnerability-free contracts are very likely to be falsely detected. We will show in Section IV-C that the noise code can cause a lot of inaccuracies.

To address the issues, we propose smart contract bytecode oriented normalization and slicing techniques to augment bytecode matching. Specifically, we carry out normalization to uniform the diverse bytecode for matching on two aspects, i.e., data normalization by labeling the data values with meaningful tags, and instruction normalization by re-ordering the operands and ignoring insignificant instructions. Furthermore, to weaken the interference of noise code, we enforce the contract specific slicing by simulating the bytecode execution and tracking the data- and control-flows, to enhance bytecode matching on a clean code set.

Following the principle of matching based techniques, we can then encode both the seed and target contracts into quantitative and measurable forms of vectors. Note that, other than the bytecode instructions, the structural information plays an important role in contract oriented matching. A simple example in Figure 3 can illustrate the importance of structural information in matching, in which exchanging the order of two statements (Figure 3b) can fix the reentrancy vulnerability [11] in Figure 3a. To effectively introduce the structural information and avoid feature engineering as done in existing studies [5], [6], we design an end-to-end graph embedding algorithm to encode the slice-based control flow graphs into vectors and measure the vector similarities for bytecode matching. We leverage the measurement results to report potentially vulnerable contracts if their vectors are highly similar to those known vulnerable ones.

We implement a prototype and evaluate it on over two million closed-source smart contracts and about 32.5k open source contracts. The results show that our method can effectively hunt hundreds of vulnerabilities using just a few known vulnerable contracts as the seeds with high performance and precision. In total, we detect 160 exploitable vulnerabilities that are verified one by one with specially developed attack payloads in a private blockchain environment. We have also demonstrated the necessity of normalization and slicing and the effectiveness of considering the structural information. Our comparative analysis shows that our method generally outperforms state-of-the-art detectors.

It is also worth noting that we identify 23 honeypot contracts [12] during an early stage of auditing because those honeypots were disguised as vulnerable to lure the attackers. Using the uncovered honeypot contracts as seeds, we further discover 33 other honeypots via the same matching method. Their harmfulness is manually verified as well.

This paper makes the following major contributions.

- To the best of our knowledge, we are the first to introduce the graph embedding approach in smart contract bytecode matching to finding unknown vulnerabilities. We propose normalization and slicing to address two contracts specific problems.
- We implement a prototype and evaluate it on real-world contracts. The experiments show its scalability (0.47 seconds per contract), high precision (91.95% for top-ranked) and effectiveness (160 true positives and higher accuracy in comparative analysis).
- We also use the method to successfully detect dozens of honeypot contracts, further illustrating the capability of our method.

The rest of this paper is organized as follows. Section II briefly describes some background and five kinds of vulnerabilities we aim to detect. Section III elaborates the approach details and Section IV presents the evaluation results. We

discuss limitations and future directions in Section V and related work in Section VI. Section VII concludes the paper.

II. Background

In this section, we provide a concise background of our work including the data processing mechanism of the Ethereum Virtual Machine (EVM) and smart contract vulnerabilities we focus on in this paper.

A. EVM Basics

Smart contracts are compiled into bytecode and executed in EVM. During an execution, EVM maintains a runtime *stack*, a transient *memory* and a persistent *storage*. Typically, upon an invocation, inputs are loaded to the stack and EVM executes the instructions to operate the data residing the stack, memory and storage.

The stack acts as the registers in a typical x86 machine and most instructions can only operate data in the stack. For example, an `ADD` instruction pops two elements from the stack, computes their sum and pushes the result back to the stack. The memory maintains volatile data at run time, usually those with size larger than 32 bytes. `SHA3` computes the hash value for a chunk of data in the memory. `MLOAD` and `MSTORE` are used to interact between the stack and the memory. Data in the stack or the memory are discarded as an execution terminates. The storage hosts data that are saved in the blockchain and can span their life across different executions. In other words, the states of a contract (e.g., the balance) and the global variables (i.e., the field variables in a contract) are kept in the storage. Instructions `SLOAD` and `SSTORE` exchange data between the stack and the storage. Apart from the memory chunks, the typical and also maximum size of an data element in the stack and the storage is 256-bit.

In addition to the above data regions and operations, a smart contract may terminate abnormally if certain condition is not satisfied. In such cases, EVM reverts the storage with instructions like `REVERT`. When an execution ends normally at `RETURN`, all changes are preserved.

B. Smart Contract Vulnerabilities

Next we will present five kinds of smart contract vulnerabilities we are interested in. All these vulnerabilities result from inappropriate handling of dirty and untrusted data, which have been reported to cause tremendous losses [1], [2], [13].

Integer overflow. The function is Figure 2a transfers the caller's balance to multiple recipients. Line 6 ensures the caller possesses sufficient amount of tokens. However, a preceding multiplication at line 3 suffers from an integer overflow. An attack providing two receivers and a *value* of 2^{255} can emit an overflow and then an *amount* of *zero*, which passes the requirement check at line 6. Eventually, the sender pays nothing but the recipients gain astronomical tokens.

Reentrancy. Figure 3a presents an example withdrawal function that involves a reentrancy vulnerability. The function first checks whether the caller's account has enough balance to withdraw, transfers the ethers and then deducts his balance. A malicious contract can exploit the vulnerability by providing a fallback function that iteratively invokes the *withdraw* function. According to the smart contract semantics, the fallback function is triggered by ether transfer via *call* at line 3. Only after the fallback function completes, can the subtraction at line 4 takes place. By this means, the attacker can exhaust the ethers held by the contract and gain more than his deposit.

Bad randomness. We illustrate the bad randomness vulnerability with a function in a lottery contract shown in Figure 4. A player is required to pay at least one ether and then depending on a hash value of the execution time, he has a chance to gain 1.9 ethers. Timestamp is used as a seed for generating random numbers, i.e., the hash value. However, an adversary understanding the logic of the contract can simulate the behaviors and thus participate in the game at the right time, making himself win for every play. We think those blockchain data that can be observed and simulated by adversaries (e.g., *timestamp*) are bad randomness sources.

```
1 function play() public payable {
2   require(msg.value >= 1 ether);
3   if (keccak256(timestamp) % 2 == 0) {
4     msg.sender.transfer(1.9 ether);
5   }
6 }
```

Fig. 4. Example for bad randomness.

Unprotected ownership is also known as the *access control* problem in DASP [13]. It corresponds to a condition in which privileged functions for changing the ownership of a smart contract is accessible to common users. Consequently, any user can become the owner and perform privileged operations, e.g., burning new tokens or destructing the contract. Below is an example from DASP [13].

```
1 function initContract() public {owner = msg.sender;}
```

Mishandled exceptions. Typically, a callee function in another contract may raise an exception, revert the state of the callee contract and then return *false* indicating a failing execution [11]. However, the exception may not be propagated to the caller. Without explicitly checking the return value before further operations can lead to unexpected behaviors. The example code below implements ether transfer via *send* and carries out balance subtraction before the action, so as to avoid reentrancy attacks. Unfortunately, without checking whether the transfer action has been conducted properly or not, the contract owner may observe inconsistent *balances*.

```
1 balances[msg.sender] -= _amount;
2 msg.sender.send(_amount);
```

III. Methodology

We propose an approach to detecting smart contracts vulnerabilities. The high level idea is to *extract patterns from smart contracts and inspect whether any unknown patterns can match vulnerable patterns*.

Figure 5 presents the overflow of our method. For a given smart contract, either unknown or with known vulnerabilities, we preprocess the bytecode and build the control flow graph (CFG), based on which we pinpoint the slice criteria and extract the corresponding slices on top of bytecode execution

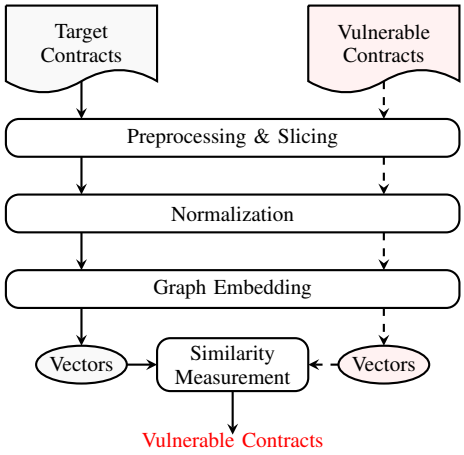


Fig. 5. A high-level overview of our method.

simulation. Then we normalize the slices and later employ a graph embedding network to vectorize the slice-oriented control flow graphs. At last, we test if vectors denoting the slices from target contracts can match any vectors for vulnerable slices by measuring the pair-wised similarities and report potentially vulnerable contracts with high similarities. We will detail these steps in below sections.

A. Preprocessing and Slicing

The smart contract bytecode does not directly carry operand information along with the instructions. Instead, all instructions, except the PUSH family, obtain their operands from the runtime stack. Without the associated stack, we cannot know the destination of a JUMP instruction or if either operand of an ADD instruction originates from the input. As done in many other methods [11], [14], we simulate the bytecode execution and construct the CFGs. Along with the simulation, we track the data- and control-flows and perform slicing according to specialized slice criteria.

1) Slice Criteria: Look back the vulnerabilities of interest in Section II. We think that all data from outside of smart contracts, including those on the blockchain (e.g., *timestamp* and user balances), have a chance to be controlled by adversaries. Improper operations on such data are prone to exploitations. Therefore, we extract the instruction sequences correlated to such data as potentially vulnerable behaviors.

Specifically, we decide the following four types of slice criteria. First, we care the *transaction data* provided by the users with an invocation. The instructions such as CALLDATALOAD, CALLER, CALLVALUE load transaction data from the input for later operations. Second, *block data* that can be retrieved from the blockchain by BLOCKHASH, TIMESTAMP and other instructions are taken into account. Third, we think the *storage data* denoting the internal state of smart contracts play an important role. Instruction SLOAD fetches data from the storage. Last, the *return value* of an external call can also be affected by adversaries, leading to unexpected behaviors, and thus we take the correlated instructions like CALL into account.

2) Slicing: Employing a simulated execution to track the data- and control flows and understand the dependence between instructions, we leverage classic slicing algorithms [15] to extract slices starting from the instructions that introduce the aforementioned four kinds of data into the runtime stack. In particular, we traverse every reachable path to collect correlated instructions. It is worth noting that our slicing procedure differs from classic slicing algorithms in two aspects.

First, we include in slices the instructions denoting abnormal terminations, e.g., REVERT and INVALID, which, as we mentioned earlier, discard all changes in current execution but do not take any operands. Such instructions are important as they may indicate a failing assertion that indeed guarantees the safety of the execution. Without data dependent on the data D_I of interesting, they are put into a slice if the specific path reaching such an instruction leaves data derived from D_I in any data regions, e.g., the stack.

Second, though a bytecode instruction usually does not carry any explicit data, we associate the data operands with the opcode to form a node in slices. Operands carry important information in differentiating the scenarios an instruction works in. For example, ADD computes the result of two input integers in one slice but calculates the memory offset in the other slice. The instruction opcode ADD itself cannot tell much about the scenarios but the associated operands give us an opportunity of distinguish them. Eventually, a node in slices consists of the opcode and the data in the form of $ADD(op1, op2)$. Note that unknown data like the input are represented by abstract symbols.

3) Example: In Figure 6, we present the slice for *amount* in Figure 3a. We eliminate irrelevant instructions including MSTORE, SHA3 and SLOAD and obtain a concise sequence with all instructions operating data derived from the slice criterion, significantly reducing the interference of noise code.

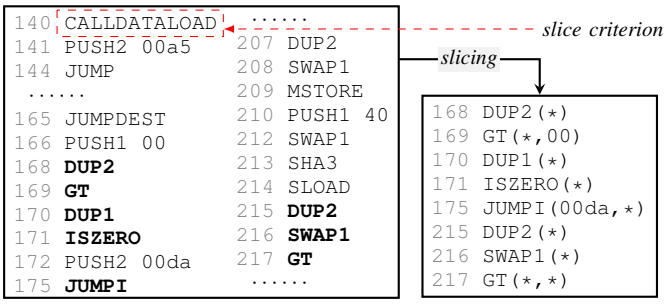


Fig. 6. An example slice (right) for CALLDATALOAD in an instruction sequence (left).

B. Normalization

Slicing eliminates irrelevant instructions, but as we mentioned previously, different compilers can result in discrepancies, obstructing bytecode matching. In this section, we will present two normalization techniques that uniform the slices for similar source code pieces as far as possible.

1) Data Normalization: We have attached the data with corresponding opcodes to form the nodes in slices. However, it is not enough to decide if an instruction in two slices actually perform very similar actions or not by inspecting only the values. Coarse-grained value types adopted by many previous

works [6], [16], [17] are neither sufficient as all values in smart contract bytecode can be viewed as 256-bit integers.

We present a fine-grained data tagging mechanism. More specifically, we tag the data introduced by the slice criteria and propagate the labels. Furthermore, when we propagate the tags from operands to the result, we use a new label to represent the operation. Table I shows part of the instructions and the corresponding tags for their results, i.e., the data inserted to the stack. For an instruction with operands (e.g., ADD), we concatenate the specified tag with the operands tags. For example, assuming two operands of ADD are tagged with *calldata* and *blk_data* separately, the resulting data will possess a tag of "*arith_res* | *calldata* | *blk_data*". As a result, the slices can carry more useful information for differentiating scenarios and associating similar usages.

TABLE I
A PORTION OF INSTRUCTIONS AND THE CORRESPONDING TAGS FOR THEIR OUTPUTS.

Instructions	Tags
CALLDATALOAD, CALLDATACOPY	*calldata*
CALLER, ORIGIN	*tx_data*
BLOCKHASH, TIMESTAMP	*blk_data*
SLOAD	*sto_data*
CALL	*call_ret*
ADD, MUL	*arith_res*
NOT, AND, OR	*bit_res*
LT, GT, SLT, EQ	*cmp_res*
MLOAD	*mem_data*
PUSH	*literal*

2) Instruction Normalization: Re-ordering the normalized tags alphabetically is commonly used in normalizations [16], which helps reduce the differences between operations on the same data. For example, in any case we should consider $ADD(calldata, literal)$ and $ADD(literal, calldata)$ indeed do the same thing. We adjust the order of tags for one operand. However, to preserve the utility, we only re-order the operands for arithmetic/bit operations ADD, MUL, AND, OR and XOR and comparison instructions EQ, LT, GT, SLT and SGT. The former, plus EQ, have no side effect after changing the positions of operands. The remaining comparison instructions can be replaced with opposite opcode if re-ordering occurs. For example, we make $ADD(calldata, literal)$ for both instructions in the above example and generate a new notation $GT(calldata, literal)$ for $LT(literal, calldata)$.

In addition, slices may contain instructions that only move or duplicate data in the stack, e.g., DUP1 and SWAP1 in Figure 6 . The DUP and SWAP families have 16 variants, i.e., DUP1 $\sim$ DUP16 and SWAP1 $\sim$ SWAP16. Depending on the compilers, same source code may be transformed to sequences with different DUP/SWAP instructions. Moreover, they are usually used with comparison instructions to construct an equivalent behavior, e.g., "SWAP, GT" v.s. "LT". Since such instructions, together with POP, are loosely correlated with the behaviors described in source code, we believe they are insignificant in code matching and thus ignore all DUP, SWAP and POP instructions from the slices.

3) Example: We show in Figure 7 an example to demonstrate the effectiveness of the proposed normalization. The first slice is the one in Figure 6 and the second one is extracted from bytecode generated by compiler v0.4.25. With the normalization techniques, we obtain the same tagged slice, which helps us quickly recognize the same behaviors within the two original slices and thus facilitates the code matching.

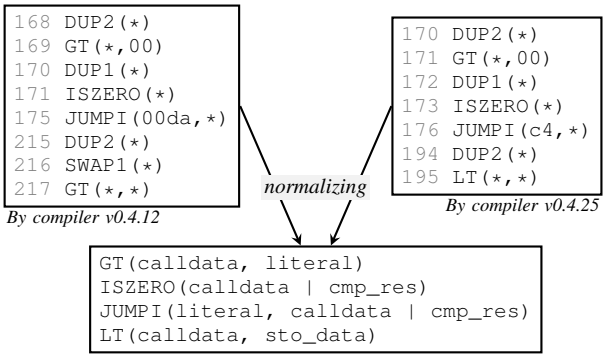


Fig. 7. An example showing that normalization eliminates the diversity resulting from different compilers for the same piece of code.

Algorithm 1: Slice Vectorization.

Input: Contract $C = \{G_1, G_2, \cdots\}$; Embedding size S
Output: Embedding vector $\mathcal{V}_{G_i} \in R^S$ for each G_i

Randomly initialize $\mathcal{V}_{G_i}$ for each $G_i \in C$;
foreach $G_i \in C$ **do**
  $SG_n = SubGraph(n, G_i)$ for each $n \in N_i$;
end
foreach $G_i \in C$ **do**
  $J(\lambda, P) = -\log \sum_{n \in N_i} \Pr(SG_n | \mathcal{V}_{G_i})$;
  $\lambda = \lambda - \alpha \cdot \frac{\partial J}{\partial \lambda}$;
  $P = P - \alpha \cdot \frac{\partial J}{\partial P}$;
end
foreach $G_i \in C$ **do**
  $J(\mathcal{V}_{G_i}) = -\log \sum_{n \in N_i} \Pr(SG_n | \mathcal{V}_{G_i})$;
  $\mathcal{V}_{G_i} = \mathcal{V}_{G_i} - \alpha \cdot \frac{\partial J}{\partial \mathcal{V}_{G_i}}$;
end

C. Graph Embedding

We have obtained uniform slices and aim to embed them into a vector space for similarity measurement. In order to capture more structural information, we convert each slice into a control flow graph and refer to graph2vec [18] for end-to-end graph embedding, instead of vectorizing each node and then aggregating the node vectors into a graph vector as done by Gemini [5]. graph2vec is an unsupervised graph embedding technique which learns the representations of whole graphs. The main idea is to treat the rooted subgraph around a node in a graph as a word and the graph as a document. The learning target is to maximize the likelihood of a rooted subgraph extracted from a graph to be occurring in the graph. By following a document embedding process, graph2vec iteratively learns the corresponding graph's representation. Theoretically, graph2vec can capture more subtle structural information than linear substructure based graph embedding techniques [19].

In this study, viewing each instruction as a node and a slice as the graph, algorithm 1 depicts how the slices $\{G_i\}$ are embedded into vectors $\{\mathcal{V}_{G_i}\}$ per smart contract C. The key of the algorithm is to update the embeddings of the rooted

subgraphs and the whole graph, such that the learned vector of a slice should be close to its rooted subgraph vectors but far away from other ones as far as possible.

In the algorithm, a rooted subgraph consists of an instruction in the slice and its neighbors, and its corresponding embedding is computed with eq. 1,

$$SubGraph(n, G_i) = \tanh\left(v_n + P \sum_{m \in Neib(n,G_i)} v_m \right) \quad (1)$$

in which P is the parameter matrix and $Neib(n, G_i)$ denotes the neighbor nodes of $n \in N_i$. N_i is the set of all nodes in G_i. A function λ maps the combination of an instruction n and its associated data labels to an S-dimensional vector v_n.

The likelihood of a rooted subgraph SG_n to be occurring in a slice G_i is computed with eq. 2.

$$\Pr\left(SG_n | \mathcal{V}_{G_i}\right) = \frac{\exp\left(\mathcal{V}_{G_i} \cdot SG_n\right)}{\sum_{w \in N_C} \exp\left(\mathcal{V}_{G_i} \cdot SG_w\right)} \quad (2)$$

Here, we denote all nodes in all slices as N_C.

Both λ and P are trained in the second stage (lines 5∼9). We then keep them fixed and iteratively train the slices' embeddings (lines 10∼13). The learning rate α is empirically set to 0.001 as done in [18]. Finally, we obtain the S-dimensional (64-dimensional in this study) vector representations for the slices, which can effectively maintain the structural information of the slices.

D. Vulnerable Smart Contract Detection

After graph embedding, for every slice, either extracted from a target contract or reflecting a known vulnerability, a corresponding vector is generated. For a given vulnerable contract, among its candidate slices, we mark only the slice that best embodies the logic of the relevant vulnerability as vulnerable. We use $\mathcal{V}_t$ to indicate a vector for a target slice and $\mathcal{V}_k$ for a vulnerable one. As same as done in other matching-based studies [4]–[6], [20], we then calculate the *cosine distance* between each $\mathcal{V}_t$ and every $\mathcal{V}_k$ to measure their similarity. The larger the value, the higher the similarity between the two slices. We rank the results based on the similarities and eliminate those pairs with relatively low similarities based on a threshold. Section IV-B2 will show how we choose a proper threshold. Those pairs with greater similarities are reported and the involved target slices are marked potentially vulnerable. The corresponding target contracts are further manually audited to confirm the existence of vulnerabilities.

IV. Evaluation

In this section, we will evaluate our method with smart contracts deployed on Ethereum. We will first describe experiment setup and then present the results, following which we discuss the comparative analysis and show some case studies. At last, we talk about the honeypot contracts that were uncovered by our method.

A. Experiment Setup

We implement a prototype system in C++ and Python. Our evaluation is carried out on an Ubuntu 16.04 (x64) machine with Intel Core CPUs (3.20 GHz × 4) and 8 GB memory.

TABLE II
DATASETS INFORMATION FOR EVALUATION.

Dataset	#Contracts	Description
Dataset I	2,297,058	Smart contracts in bytecode
Dataset II	32,499	Open-source smart contracts
Dataset III	24	Known vulnerable contracts from CVE

We construct three datasets from different sources, as shown in Table II. Dataset I contains over two million smart contracts we extracted from the Ethereum blockchain by Mist [21]. All those contracts are in the form of bytecode, whereas Dataset II includes the open-source smart contracts we collected from Etherscan [22]. We built the last one, Dataset III, by screening out the representative and suitable ones from CVE hosted vulnerable smart contracts. More specifically, among the 24, there are 10 about integer overflow, 4 for reentrancy, 6 for bad randomness, 2 for unprotected ownership and the rest for mishandled exceptions.

B. Experiment Results

1) Performance: Our method embodies two major stages, data preparation and vulnerability detection. The former one consists of program slicing, normalization and graph embedding. Since slicing and normalization are interleaved during the bytecode analysis, we treat them as a whole for evaluating the performance.

TABLE III
PERFORMANCE FOR 2,297,058 CLOSED-SOURCE SMART CONTRACTS IN DATASET I.

Stage		Total (s)	Avg. (s)	Std.
Data Preparation	Slicing/Norm.	204,739	0.089	0.531
	Embedding	849,710	0.370	0.187
	Total	1,054,449	0.459	0.561
Vulnerability Detection		28,654	0.012	0.013

In Table III, we list the total and average time cost for each procedure on Dataset I. The last column (Std.) shows the standard deviations. We can see that due to the large size of Dataset I, preparing the data takes more than twelve days. However, for each single smart contracts, the time cost is only about 0.46 seconds. With the prepared data, we leverage the critical slices from the smart contracts in Dataset III to detect vulnerable contracts in Dataset I by calculating the similarities between each target contract and known vulnerable ones. For each one in Dataset I, the average time cost is as low as 0.012 seconds, showing that the similarity calculation based on vectors is highly efficient. In summary, the average time for one target smart contract is about 0.47 seconds, demonstrating high performance for large-scale vulnerability detection based on bytecode matching.

2) Accuracy: In this section, we will present the accuracy aspect of vulnerability detection by inspecting the false positive (FP) rate and false negative (FN) rate. Before that, we tune the parameters with experiments on a subset of the smart contracts.

Parameter tuning. We first tune the similarity threshold value. We select 3,000 smart contracts from Dataset II for vulnerability detection and use the vulnerable ones in Dataset III as the seeds for matching. We conduct the detection and count the false alarms. Besides, we sampled 300 target contracts for manual auditing, the results of which are considered for the false negative rates.

TABLE IV
RESULTS FOR DIFFERENT SIMILARITY THRESHOLDS.

Threshold	FP Rate	FN Rate
100%	0%	71.43%
95%	13.51%	42.86%
90%	14.29%	14.29%
85%	56.10%	14.29%
80%	74.10%	14.29%

We obtain the results for several typical threshold values (80% $\sim$ 100% with a step of 5%) and show them in Table IV. When the threshold is decreased from 90% to 85%, the FP rate increases drastically while the FN rate does not change. However, if we increase the threshold from 90% to 95%, the FP rate shows slight decrement but the FN rate is substantially changed. To trade off FP and FN, we pick 90% for all subsequent experiments.

TABLE V
RESULTS FOR DIFFERENT EMBEDDING SIZES. THE LAST ROW SHOWS THE NUMBER OF EXPLOITABLE CONTRACTS OVER THE TOTAL REPORTED NUMBER, I.E., TP/TOTAL.

	S=32	S=64	S=128	S=256
Time Cost (s)	995.57	1,342.66	2,056.74	4,425.29
Detection Results	1/23	4/4	1/8	4/10

We also investigate the effect of different embedding size S. We randomly select 1,000 smart contracts from Dataset II and carry out experiments with different S. Table V lists the results. Because on average, smart contracts in Dataset II are relatively more complicated than those in Dataset I, they cost more time compared with that in Table III. In addition to the time cost, we present the number of smart contracts with exploitable vulnerabilities over the number of all reported ones. In other words, the fractions denote the precision under each setting. In order to balance the efficiency and accuracy, we choose 64 as the embedding size.

Detection Results. With the above parameters, the prototype system reports 1,220 vulnerable contracts as candidates. The numbers of contracts corresponding to integer overflow, reentrancy, bad randomness, unprotected ownership and mishandled exceptions are 732, 129, 23, 182 and 154, respectively. We deploy these smart contracts in a private blockchain to verify the results. Unfortunately, without the source code, it is highly time-consuming and unreliable to manually inspect low-level bytecode [23], which requires us to speculate the behaviors of the instructions and infer the vulnerability occurrences. Therefore, instead of examining all reported issues, we inspect the ones with top ten similarities and observe eight successful exploitations.

TABLE VI
DETECTION RESULTS FOR DATASET I AND DATASET II ON THE ASPECT OF DIFFERENT VULNERABILITIES, INTEGER OVERFLOW (IO), REENTRANCY (RE), BAD RANDOMNESS (BR), UNEXPECTED OWNERSHIP (UO) AND MISHANDLED EXCEPTION (ME).

Dataset	IO	RE	BR	UO	ME	Total
Dataset I	732	129	23	182	154	1,220
Dataset II	104/112 (92.9%)	23/23 (100%)	10/10 (100%)	11/13 (84.6%)	4/6 (66.7%)	152/164 (92.7%)

Our evaluation on Dataset II reports 164 vulnerable smart contracts, 152 among which are verified to be exploitable, 104, 23, 10, 11 and 4 for the aforementioned vulnerability types separately. We successfully verify all of them in a blockchain.

We show the results in Table VI. For Dataset II, we also list the number of exploitable contracts and the total number for each vulnerability type. The overall verified precision is 91.95% ($= \frac{8+152}{10+164}$).

Before CVE stopped accepting smart contract vulnerabilities, we have been granted three CVE IDs for the reported vulnerabilities, CVE-2018-17882, CVE-2018-19460 and CVE-2018-19852.

C. Comparative Analysis

For better demonstrating the effectiveness of our approach, we conducted a series of comparative analysis, including comparisons with a Bag-of-Feature based method, with state-of-the-art bug-finders and with weakened version of our method, e.g., eliminating slicing or normalization. Due to the fact that the source code can greatly help analysts audit reported vulnerabilities, we perform the experiments on the smart contracts in Dataset II.

Comparison with a structure-ignorant method. Without considering the structural information, we can treat each instruction as a dimension of features and embed the slices in a way of Bag-of-Features (BoF). We implement a BoF based detection by replacing the graph embedding stage with a BoF-embedding and mapping the number of instructions to their specific dimensions. Later, we can carry out the same similarity measurement on those vectors.

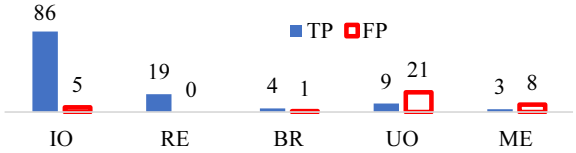


Fig. 8. Results of a structure-ignorant BoF method.

We present in Figure 8 the results of the above BoF based detection on Dataset II. For all kinds of vulnerabilities, the BoF method reports fewer true positives (TP) and for the last two types, it exhibits much higher FP rate.

We investigate the cause and find that the BoF method considers the features from slices independent of each other. However, our graph-embedding based method can capture the relations between the features, and thus better describe the characteristics of the slices, achieving higher accuracy.

Comparison with state-of-the-art detectors. We also compare our approach with four state-of-the-art tools, Oyente [11], Mythril [24], Manticore [25] and Securify v2.0 [26]. Here, we concentrate only on the most harmful vulnerabilities, i.e., integer overflow and reentrancy, and show the comparison results in Table VII.

TABLE VII
COMPARISON RESULTS WITH STATE-OF-THE-ART DETECTORS.

	Oyente	Mythril		Manticore		Securify	Our Tool	
	RE	IO	RE	IO	RE	RE	IO	RE
#Reports	172	53	122	16	22	23	112	23
#FPs	154	7	116	0	0	0	8	0
FP Rate	89.5%	70.3%		0%		0%	5.93%	

We compare with Oyente on detecting reentrancy vulnerabilities. For smart contracts in Dataset II, Oyente reports 172 reentrancy vulnerabilities in total, taking about 60 hours. Our manual verification shows that only 18 contracts are truly vulnerable. Namely, the FP rate is 89.53%, much higher than our method.

```
1 function refund() stopInEmergency {
2   if (getState() != State.Refunding) throw;
3   address investor = msg.sender;
4   if (balances[investor] == 0) throw;
5   uint amount = balances[investor];
6   delete balances[investor];
7   if (!(investor.call.value(amount)())) throw;
8   Refunded(investor, amount);
9 }
```

Fig. 9. PreICOProxy [27] where Oyente falsely reports a reentrancy.

We present one FP by Oyente in Figure 9. Instead of decreasing the amount from corresponding account, this smart contract directly deletes the account from its record (line 6) before withdrawing the balance at line 7. A reentrant invocation is thus prevented at line 4, leading to no reentrancy problems. Oyente does not recognize the behaviors for line 6 and incorrectly reports a vulnerability. Many other FPs are emitted by Oyente for similar reasons, indicating the difficulty of manually summarizing precise rules. On the contrary, our approach recognizes vulnerability patterns from vulnerable smart contracts and guarantees the presence of vulnerabilities in the extracted patterns, leading to lower FP rate.

We compare with Mythril on integer overflow and reentrancy. For smart contracts in Dataset II, Mythril detects 53 integer overflow vulnerabilities and 122 reentrancy problems, among which 7 and 116 are FPs, respectively. Our method exhibits higher precision and recall. This comparison further demonstrates the incompleteness of collecting detection rules by human beings.

Manticore [25] employs some heavyweight analysis techniques, e.g., dynamic symbolic execution. Analyzing smart contracts with Manticore is an expensive task. Our experience has shown that Manticore cannot complete the analysis or produce any valuable results for more than 70% of the vulnerable contracts within 90 minutes. To avoid non-terminable evaluations, we set a timeout of 90 minutes per contract, as done in [25]. Due to its high time overhead and the severity of false negatives, we evaluate Manticore on the 135 potentially vulnerable smart contracts reported by our tools, including 112 integer overflows and 23 reentrancy. Manticore misses 88 (84.6%) confirmed integer overflows and one reentrancy. Securify v2.0 does not detect integer overflows and thus we can only evaluate it on the 23 vulnerable smart contracts with reentrancy vulnerabilities. It shows no FPs or FNs.

Manticore works poorly in detecting integer overflows that often involve subtle logics and Securify v2.0 does not detect this type of vulnerability, though they both have good performance for reentrancy vulnerabilities with clear logic. We argue that there is lack of enough prior knowledge, i.e., detection rules, for the integer overflow. In fact, extracting and inferring complete knowledge for subtle vulnerabilities is a very difficult mission, if not impossible. In this study, the known vulnerabilities are directly employed as detection signatures. As a result, we can get valued results more quickly and avoid tedious manual work.

Comparison with underpowered approaches. Our method is powered with normalization and slicing. Below, we will eliminate either capability to evaluate their necessity for precise vulnerability detection. As shown in the first column in Table VIII, except the full solution (*N/S*), we build three underpowered tools with the following capabilities: (1) enabling normalization only (*N-only*), (2) enabling slicing only (*S-only*) and (3) disabling both techniques (*No-N/S*).

TABLE VIII
COMPARISON RESULTS WITH UNDERPOWERED APPROACHES (N/S FOR THE FULL-POWERED APPROACH AND NUMBERS FOR TP RATE IN EACH CELL).

	IO	RE	BR	UO	ME
N/S	104/112 (92.9%)	23/23 (100%)	10/10 (100%)	11/13 (84.6%)	4/6 (66.7%)
N-only	78/137 (56.9%)	16/24 (66.7%)	9/29 (31.0%)	10/34 (29.4%)	3/11 (27.3%)
S-only	76/84 (90.5%)	22/25 (88.0%)	3/3 (100%)	2/2 (100%)	3/8 (37.5%)
No-N/S	0/14560	0/2278	0/12059	0/9384	0/230

We conduct the experiments and show the results in Table VIII. For *No-N/S*, due to the large number of reported contracts, we only audit the top 20 smart contracts and count the vulnerable. From the results, we can see that our method with normalization and slicing (*N/S*) outperforms all the other three approaches. *N-only* does not apply slicing techniques and thus takes the noise into account, producing a lot of FPs. *S-Only*, on the other hand, does not resolve the differences caused by compilers, leaving a number of FNs. Directly analyzing the bytecode without any preprocessing, *No-N/S* reports too many FPs and burdens the analysts. Therefore, we can claim that, normalization and slicing adopted in our approach can effectively lower the impact of compiler diversity and noise code and emit higher accuracy for matching based vulnerability detection.

D. Case Studies

Below we will present three cases, one open source, one closed-source and the last illustrating the effect of considering the structural information.

Case 1: BattleToken. Figure 10 shows a vulnerable smart contract, BattleToken [28], which implements an Ethereum-based game. The *batchTransfer* function allows the caller to distribute his balance to a set of recipients. The developers employ a *safeSub* function to ensure the subtraction from the caller's account (line 2), reverting for insufficient balance. Similarly, *safeAdd* is used to guarantee the safety of addition for each recipient at line 5.

```
1 function batchTransfer(address[] to, uint value)
      public {
2   balances[msg.sender] = safeSub(
3       balances[msg.sender], to.length * value);
4   for (uint i = 0; i < to.length; i++) {
5     balances[to[i]] = safeAdd(balances[to[i]],
6                               value);
7   }
8 }
```

Fig. 10. Contract BattleToken [28] (CVE-2018-17882).

However, the developers perform an unprotected multiplication *to.length* * *value* at line 3 to calculate the total amount the caller needs to pay, which is prone to an integer overflow exploitation. Attackers can construct a very large *value* and make a very small number for the multiplication, leading to little decrement of the sender but huge increment of the recipients.

We discover this vulnerable smart contract using the well-known BecToken [9] as the seed with the similarity of 95.5%. However, at the source code level, the two contracts do not look the same. Neither do they look highly similar at the bytecode level. In fact, their similarity at the instruction level is only 71.3%, much lower than the adopted similarity threshold. Fortunately, the parameter *value* in both contracts undergoes similar processing and thus comparing the corresponding slices helps the matching with a high similarity.

We validate the vulnerability in Remix [29] and it has been confirmed by the developers and CVE (CVE-2018-17882).

Case 2: A closed-source contract. In this section, we show how the closed-source candidates are manually inspected through an easy-understanding example.

We flag a contract [30] potentially vulnerable. To verify the correctness, we first need to figure out the invocation interface of the vulnerable function. By comparing the fingerprint with other contracts, we luckily find that it should be called in the same way as the *batchTransfer* function in Figure 10. Then we explore how the suspicious vulnerability can be exploited. In this example, the identified slice is very similar to the one from BecToken [9], so we make an attempt in the same manner. At last, we need to confirm the success of exploitation. Again, we find that this contract holds the interface of function *balanceOf(address)*, through which we could check the crafted balance and verify the vulnerability.

However, in most cases, the above steps can rarely be fulfilled, which significantly hinders manual inspection. Therefore, in the evaluation, we mainly focus on the results for open source smart contracts in Dataset II.

Case 3: Impact of structure. The two functions in Figure 3 have the same statements with different orders, causing one vulnerable to reentrancy but the other safe. An order-ignorant method cannot capture the difference between the two contracts due to the fact that they have the same instructions, and will emit a false alarm.

We compute the similarities between the slices for *amount* in the two functions, 100% for the BoF method in Section IV-C and 74.8% for our graph embedding method, respectively. The results clearly show that the vectors generated by graph embedding can well represent the structural characteristics of the two slices, significantly reducing the similarity and suppressing the FP.

E. Honeypot Contracts

At the early stage of our study, we found a special type of smart contracts that camouflage vulnerable contracts. Our further validation shows that such smart contracts generally provide the source code, contain some obvious vulnerability patterns like reentrancy and attract the attackers to deposit and then exploit the vulnerabilities. The attackers expect to acquire a lot of ethers but indeed leave their money in the contracts without any chance of getting back. Such smart contracts are called the *honeypot contracts* [31].

```
1  mapping (string => uint) parameters;
2  function claim_reward(uint uid, bytes32
       passcode) ... {
3    require(msg.value >= parameters["price"]);
4    require(is_passcode_correct(uid, passcode));
5    uint final_reward = get_reward(uid) + msg.value;
6    if (final_reward > parameters["price_pool"])
7      final_reward = parameters["price_pool"];
8    require(msg.sender.call.value(final_reward)());
9    parameters["price_pool"] -= final_reward;
10   /* delete the user */
11 }
```

Fig. 11. Contract HODLerParadise [32].

Figure 11 presents a honeypot contract that are marked as potentially vulnerable with SMART [33] as the seed. Looking at the source code, we are very sure that there exists a reentrancy bug and so do the attackers. Hence, the attackers pay an amount of ethers to participate in the game (omitted in the code) and later pay some fee for getting reward and quiting the game. Line 8 transfers the bonus to the caller and line 9 does the subtraction. Everything looks the same as a typical reentrancy vulnerability.

However, when we try to verify it in Remix [29], we fail to steal money from the owner. The key problem lies in lines 6 and 7, which adjust the bonus pool. While looking the same, the second 'o' in *price_pool* is actually a Greek letter that spells "omicron"[1]. Because mapping for *price_pool* does not exist in *parameters*, anyone who wants to claim the reward gets nothing from the function. But in the meantime, it steals ethers from the attacker for every reentrancy attack.

When we audited the potentially vulnerable contracts at the early stage, we totally discovered 23 honeypot contracts.

Further honeypot detection. We also use the prototype system to detect honeypot contracts using four representative

[1]Here we use different fonts to distinguish the Greek letter 'o' and English letter 'o'.

```
1  contract SmallRoulette {
2    uint private secretNumber;
3    uint public lastPlayed;
4    uint public betPrice = 0.1 ether;
5    struct Game { address player; uint number; }
6    function play(uint number) payable public {
7      require(msg.value >= betPrice && number <= 5);
8      Game game;
9      game.player = msg.sender;        overwriting
10     if (number == secretNumber) {
11       msg.sender.transfer(this.balance);
12     } // other code omitted
13   }
14 }
```

Fig. 12. Simplified contract SmallRoulette [34].

honeypot contracts discovered previously as the seeds. Finally, we find 33 suspicious honeypot contracts and all are verified to be true honeypots. Below we present an example that has 99.0% of similarity with one seed honeypot.

Figure 12 shows the simplified contract, which implements a gamble game. The participants get reward if providing a number same as the secret number. Attackers may retrieve the secret number from the blockchain storage space and then make a quick profit. However, it is indeed a honeypot. Solidity compilers before v0.5.0 allow the code to use an uninitialized pointer, e.g., *game* at line 8. But at run time, this pointer points to the 0-th slot of the contract's storage. That means, a modification at line 9 overwrites *secretNumber* with the caller's address that is always a huge number. Remember that line 10 enforces a small *number*. As a consequence, no one can satisfy the condition at line 10 and all participants, either benign or malicious, lose their money.

V. Discussion

Vulnerability Types. Although in this paper we only detect five types of vulnerabilities, our method is capable of detecting many other types of vulnerabilities resulting from improper operations on external data. But we must admit that, some types are beyond our capacity, like the *greedy contracts* introduced by Nikolić et al. [35]. A greedy contract accepts ethers but has no way to release ethers in the whole contract. This characteristic cannot be described by a single slice from one function. In future, we will explore approaches to representing the whole contract in appropriate vectors and extend the matching based method to more kinds of vulnerabilities.

False Positives. Our validation shows some false positives that are quite similar to the seed slice, but in fact they exhibit completely different functionality. For example, at the bytecode level, some random number calculations share nearly the same instruction sequence as storage writes, confusing the tool. To suppress such FPs, we may increase the degree of differentiation between the slices by providing more fine-grained data attribute types.

Verification. In this paper, we verify the dangerous contracts by deploying them on a private blockchain, providing an environment similar to the real blockchain and observing the results of exploitation. In the future, we will expand our work with some automated methods to make the verification process more reliable and efficient.

Performance Deterioration. As a learning-based method, the performance of the proposed method may be downgraded over time. For example, due to the evolution of the code patterns or compilers, current vulnerable code patterns (i.e., the vulnerable slices) may not be effective enough to catch future vulnerable code fragments. However, we believe this issue can be addressed by continuously tracking the vulnerable contracts and the patterns, with which we can make the proposed technique work well. In fact, slicing and embedding new contracts to generate matching vectors is an easy job.

VI. Related Work

In this section, we will discuss research efforts related to our work and mainly focus on two aspects, matching based methods and smart contract oriented approaches.

Matching based methods. Matching-based approaches have proven to be effective in program analysis and are widely used to detect vulnerabilities.

Yamaguchi et al. extract features such as variable declarations and function calls from the abstract syntax tree and embed those features to represent corresponding functions [3], [4]. Kim et al. proposed VUDDY [36], fingerprinting the string length and hash value of a normalized function body and utilizing the fingerprints to detect vulnerable code clones. discovRE [7] searches similar functions from a binary code base using a bunch of extracted numeric and structural features. Genius [6] extracts a set of predefined basic-block level features, e.g., number of string constants and numeric constants, to denote attributed control flow graphs (ACFGs), and generate a codebook for further encoding the ACFGs and detecting vulnerabilities via similarity matching. Gemini [5] leverages the same raw features to build ACFGs but employs a graph embedding network for encoding. While determining a proper set of features can be nontrivial, our method, in contrast, utilizes an unsupervised graph embedding network for automatic extraction and encoding.

Liu et al. [37] present αDiff that extracts intra-function features using a deep neural network without any expert knowledge and performs binary code similarity detection with inter-function and inter-module features. Zuo et al. [20] co-opt the natural language process techniques for code similarity comparison, by treating instructions as words and basic blocks as statement and mining the internal relations between them. Ding et al. propose Asm2Vec [38], which directly learns the relations and vector representations based on assembly code, without any prior knowledge required. Compared with these solutions taking the whole function body into account, our method eliminates the noise as far as possible before vector embedding to concentrate on more vulnerability-relevant operations.

Smart contract specific methods. Symbolic execution is widely adopted in smart contract analysis [39]. Oyente [11] traverses all possible paths symbolically to detect vulnerabilities including reentrancy. Maian [35] focuses on vulnerabilities across multiple execution paths. Osiris [40] detects integer operation bugs through symbolic execution and taint analysis. teEther [41] and sCompile [42] utilize symbolic execution

to explore paths with critical operations. Manticore [25] and Mythril [24] are two other tools that employ symbolic execution techniques for vulnerability detection. With manually gathered detection rules, the above symbolic execution techniques inevitably emit high FP/FN rate, as we have shown in Section IV-C.

Apart from symbolic execution, other techniques are also applied to enforce the safety of smart contracts. Securify [14] extracts semantic information from the bytecode and checks if any safety properties are violated. Vandal [43] inspects if a given smart contract violates predefined specifications after decompiling the bytecode to an intermediate representation. Zeus [44] translates the Solidity source code into LLVM bitcode and leverages the LLVM framework for abstract interpretation and symbolic model checking. VeriSmart [45] and VerX [46] employ formal verification to ensure the safety properties. ContractFuzzer [47] applies fuzzing techniques to test the smart contracts and uses predefined rules to detect vulnerabilities. ReGuard [48] converts smart contracts to semantically equivalent C++ programs and leverages fuzzing to detect vulnerabilities. These approaches also require to obtain proper rules or specifications, while our method automatically extracts the vulnerability patterns from slices and matches them with target contracts for detection.

There are also some works concentrating on honeypot contract analysis. Sanjuas presents a couple of honeypot contracts he came across, and explains the traps [31]. Torres et al. sum up several kinds of honeypots and build a tool based on some abstract features [12]. In contrast, our method can systematically detect vulnerable and honeypot contracts without summarizing any features.

VII. Conclusion

Vulnerable smart contracts on Ethereum may cause financial losses, making it a critical task to detect vulnerabilities in smart contracts. Matching-based detection by extrapolating known vulnerabilities to unknown has proven to be effective, but directly adopting the technique to smart contract analysis is obstructed by diversity of bytecode generation and interference of noise code. In this paper, we present contract bytecode oriented normalization and slicing techniques to address the above issues. Moreover, we design an unsupervised graph embedding algorithm to vectorize the code graphs of normalized slices and measure the vector similarities for bytecode matching and vulnerability detection. Our evaluations show that the method can effectively and accurately identify a considerable number of vulnerable contracts and dozens of honeypot contracts, outperforming state-of-the-art detectors.

References

[1] C. Metz, "The biggest crowdfunding project ever—the DAO—is kind of a mess," https://www.wired.com/2016/06/biggest-crowdfunding-project-ever-dao-mess.

[2] "Overflow error shuts down token trading," http://hiphopworldmagazine.com/homeposts/overflow-error-shuts-down-token-trading.

[3] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning," in *Proceedings of the 5th USENIX Workshop on Offensive Technologies*, ser. WOOT'11, 2011, pp. 118–127.

[4] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC'12, 2012, pp. 359–368.

[5] 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*, ser. CCS'17, 2017, pp. 363–376.

[6] 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*, ser. CCS'16, 2016, pp. 480–491.

[7] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: efficient cross-architecture identification of bugs in binary code," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, ser. NDSS'16, 2016, pp. 1–15.

[8] M. Fröwis and R. Böhme, "In code we trust? - measuring the control flow immutability of all smart contracts deployed on ethereum," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, ser. DPM'17, CBT'17. Springer, 2017, pp. 357–372.

[9] "BecToken," https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d.

[10] "Beercoin," https://etherscan.io/address/0x7367a68039d4704f30bfbf6d948020c3b07dfc59.

[11] 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*, ser. CCS'16, 2016, pp. 254–269.

[12] C. F. Torres, M. Steichen, and R. State, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *Proceedings of the 28th USENIX Security Symposium*, ser. USENIX Security'19, Aug. 2019, pp. 1591–1607.

[13] "DASP top 10," https://www.dasp.co/.

[14] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS'18, 2018, p. 67–82.

[15] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," in *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ser. SDE'84, 1984, pp. 177–184.

[16] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai, "AntMiner: mining more bugs by reducing noise interference," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE'16, 2016, pp. 333–344.

[17] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE'05, 2005, pp. 306–315.

[18] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," *arXiv preprint arXiv:1707.05005*, 2017.

[19] B. Adhikari, Y. Zhang, N. Ramakrishnan, and B. A. Prakash, "Distributed representations of subgraphs," in *Proceedings of 2017 IEEE International Conference on Data Mining Workshops*, ser. ICDMW'17, 2017, pp. 111–117.

[20] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Proceedings of the 2019 Network and Distributed Systems Security Symposium*, ser. NDSS'19, 2019, pp. 1–15.

[21] The Ethereum Community, "Mist - browse and use dapps on the ethereum network," https://github.com/ethereum/mist.

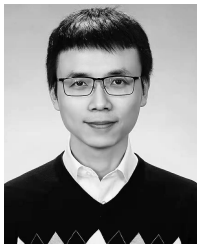
[22] Etherscan, "Ethereum (ETH) blockchain explorer," https://etherscan.io/.

[23] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, Oct. 2018.

[24] ConsenSys, "Mythril - security analysis tool for EVM bytecode," https://github.com/ConsenSys/mythril.

[25] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'19, 2019, pp. 1186–1189.

[26] "Securify v2.0," https://github.com/eth-sri/securify2.
[27] "PreICOProxy," https://etherscan.io/address/0x88466183e69c8f681bdeb538d57cf270255f8762.
[28] "BattleToken," https://etherscan.io/address/0x4daa9dc438a77bd59e8a43c6d46cbfe84cd04255.
[29] The Ethereum Community, "Remix - the solidity ide," http://remix.ethereum.org.
[30] "A closed-source contract," https://etherscan.io/address/0xddc36be766d4328dbf898cfd9826478659dba5c3.
[31] J. Sanjuas, "An analysis of a couple ethereum honeypot contracts," https://medium.com/coinmonks/an-analysis-of-a-couple-ethereum-honeypot-contracts-5c07c95b0a8d, 2018.
[32] "HODLerParadise," https://etherscan.io/address/0xc03b0dbd201ee426d907e367f996706cf53b8028.
[33] "SMART," https://etherscan.io/address/0x60be37dacb94748a12208a7ff298f6112365e31f.
[34] "SmallRoulette," https://etherscan.io/address/0x01B21934Ba28DfD8a22c4D21c710290500A5081F.
[35] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC'18, 2018, pp. 653–663.
[36] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, ser. S&P'17, 2017, pp. 595–614.
[37] 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*, ser. ASE'18, 2018, pp. 667–678.
[38] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, ser. S&P'19, 2019, pp. 472–489.
[39] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and tools for the static analysis of ethereum smart contracts," in *Proceedings of the 30th International Conference on Computer Aided Verification*, ser. CAV'18, 2018, pp. 51–78.
[40] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC'18, 2018, pp. 664–676.
[41] J. Krupp and C. Rossow, "teEther: Gnawing at ethereum to automatically exploit smart contracts," in *Proceedings of the 27th USENIX Security Symposium*, ser. USENIX Security'18, Aug. 2018, pp. 1317–1333.
[42] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang, "sCompile: Critical path identification and analysis for smart contracts," in *Formal Methods and Software Engineering*. Springer, 2019, pp. 286–304.
[43] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *CoRR*, vol. abs/1809.03981, 2018.
[44] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Proceedings of 25th Annual Network and Distributed System Security Symposium*, ser. NDSS'18, 2018, pp. 1–15.
[45] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *Proceedings of the 41st IEEE Symposium on Security and Privacy*, ser. S&P'20, 2020, pp. 1678–1694.
[46] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *Proceedings of the 41st IEEE Symposium on Security and Privacy*, ser. S&P'20, 2020, pp. 1661–1677.
[47] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE'18, 2018, pp. 259–269.
[48] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: Finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE'18, 2018, pp. 65–68.



Jianjun Huang received the Ph.D. degree in Computer Science from Purdue University. He is currently an assistant professor at School of Information, Renmin University of China. His research interests focus on program analysis, vulnerability detection, mobile security and blockchain security.

Songming Han received the M.S. degree in Information Security from Renmin University of China. His research interests focus on software security analysis and blockchain security.

Wei You received the Ph.D. degree in Computer Science from School of Information, Renmin University of China. He is currently an associate professor at School of Information, Renmin University of China. His research interests focus on program analysis, mobile security and Web security.

Wenchang Shi received the Ph.D. degree in Computer Science from Institute of Software, Chinese Academy of Sciences. He is currently a professor at School of Information, Renmin University of China. His research interests focus on information security, trusted computing, cloud computing, computer forensics and operating systems.

Bin Liang received the Ph.D. degree in Computer Science from Institute of Software, Chinese Academy of Sciences. He is currently a professor at School of Information, Renmin University of China. His research interests focus on program analysis, vulnerability detection, mobile security and AI security.

Jingzheng Wu received his Ph.D. degree in computer software and theory from the Institute of Software, Chinese Academy of Sciences, Beijing, in 2012. He is an associate research professor at the Institute of Software, Chinese Academy of Sciences, Beijing. His primary research interests include system security, vulnerability detection and covert channels.

Yanjun Wu received his Ph.D. degree in computer software and theory from the Institute of Software, Chinese Academy of Sciences, Beijing, in 2006. He is a research professor at the Institute of Software, Chinese Academy of Sciences, Beijing. His primary research interests include operating system and system security.