# Ethereum Smart Contract Representation Learning for Robust Bytecode-Level Similarity Detection

Zhenzhou Tian[1,2,3*], Yaqian Huang[1,2], Jie Tian[1,2], Zhongmin Wang[1,2], Yanping Chen[1,2], and Lingwei Chen[4*]
[1]*School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an, 710121, China*
[2]*Key Laboratory of Network Data Analysis and Intelligent Processing, Xi'an, 710121, China*
[3]*Xi'an Key Laboratory of Big Data and Intelligent Computing, Xi'an 710121, China*
[4]*Department of Computer Science and Engineering, Wright State University, Dayton, OH, USA*

***Abstract*—Smart contracts are programs that run on a blockchain, where Ethereum is one of the most popular ones supporting them. Due to the fact that they are immutable, it is essential to design smart contracts bug-free before they are deployed. However, various defects have been found in the deployed smart contracts, causing huge economic losses and lowing people's trust. Writing secure smart contracts is far from trivial, where developers tend to engage in reliable resources or social coding platforms to reuse code. This leads to a large number of similar contracts with potential security risks. Therefore, detecting similarity of smart contracts helps to avoid vulnerabilities, identify threats, and improve the security of Ethereum. In this paper, we design a learning-effective and cost-efficient model, called SmartSD, for Ethereum smart contract similarity detection. Different from the current research efforts, SmartSD is performed on a bytecode level and leverages deep neural networks to learn the latent representations from the opcode sequences for smart contract bytecodes, where the representation learning and similarity measurement are supervised via siamese neural networks. The experimental evaluations demonstrate that SmartSD outperforms EClone's 93.27% accuracy, achieving 98.37% high detection accuracy and 0.9850 F1-score, which is computationally tractable and effectively mitigates the interference caused by compilers.**

***Index Terms*—Smart contract, Similarity detection, Deep learning, Siamese neural network**

## I. Introduction

[1] Smart contracts are Turing-complete programs that run on a blockchain. Among the emerging blockchain platforms, Ethereum [27] is the most popular one to operate smart contracts. Different from the traditional programs, smart contracts have the unique characteristics of openness, transparency, non-tamperability, and independence from third parties. In other words, they can be integrated to different decentralized applications (e.g., financial services and supply chain management) after deployment; more importantly, they are immutable to be modified or patched, where the only way to change a smart contract is to deploy a new instance. It is hence essential to design smart contracts bug-free before their deployments [2]. Unfortunately, various defects have been found in the deployed smart contracts [3], [11], [25]. To put it into perspective, according to a recent study, 97% of the collected contracts are annotated as vulnerable by one or more vulnerability analysis tools [5]. This enables attackers to exploit the security vulnerabilities raised by these defects to fulfill their economic intents, and cause huge economic losses [19], which has also severely lowered people's trust in Ethereum smart contracts.

Smart contracts are usually written in high-level programming languages, where Solidity [20] is the most widely used one. With high-level languages facilitating smart contract developments, writing secure smart contracts is still not easy. The major factor behind this is that these programming languages are young and evolving, where every change of smart contracts would introduce significant updates into the languages. Developers may need many efforts to get familiar with newer versions, and thus tend to frequently engage in reliable resources (e.g., Etherscan) or social coding platforms (e.g., GitHub) for code reuse to expedite smart code development process. Considering the widespread defects existing in the deployed smart contracts, this naturally leads to a large number of similar smart contract codes with potential security risks. With this in mind, detecting the similarity of smart contracts may allow developers to avoid using the compromised ones; by comparing contracts with known vulnerabilities, it helps to identify threats and improve Ethereum security.

Similarity detection is a long-term research topic, but the investigation into smart contracts has been scarce, especially for similarity detection on Ethereum smart contract bytecodes. The deployment of a smart contract proceeds by first compiling its source code into EVM bytecode, encapsulating bytecode into a transaction, and then sending the transaction to the zero address [26]. On one hand, smart contract source codes are not always accessible to avoid attacks, while only their bytecodes are deployed to run on EVM. On the other hand, many smart contract defects occur after being compiled as bytecodes [21]. To this end, our research goal here is to automatically identify the similarities over the Ethereum smart contract bytecodes. To achieve this goal, we face a challenge: developers may compile smart contracts using different compilers or the same compiler with different optimization options enabled; accordingly, these cross-compiler and cross-optimization-level compilations will impose discrepancies on opcode distributions of the resulting bytecodes, even when their source codes are exactly the same. Direct fingerprint generation on bytecodes may not be a good idea to capture their semantics. It needs a better formulation to learn the

* Corresponding: tianzhenzhou@xupt.edu.cn and lgchen@mix.wvu.edu
[1]DOI reference number: 10.18293/SEKE2022-040

higher-level representations of contracts and automatically detect the similarity among them.

To address the above challenge, in this paper, we design a learning-effective and cost-efficient model, called *SmartSD*, for Ethereum **Smart** contract **S**imilarity **D**etection. Different from the limited research efforts that build upon either fingerprints susceptible to interference [10], cost-expensive runtime traces [14], [15], or source code structure [6], our model first decomposes the given bytecode into EVM instructions and abstract them as an opcode sequence, and then elaborate deep neural network structure to learn the representation encoding structures and semantics of the opcode sequence. In order to supervise the representation learning and similarity measurement, SmartSD further builds up siamese neural networks (SNN) [12] to train the model, such that the model can be successfully applied to the validation and test data sets. The major traits of our work can be summarized as follows:

- We investigate smart contract similarity detection on the bytecode level, leveraging feature learning ability of deep neural networks to learn latent representation from the abstracted opcode sequence for each bytecode. The proposed method is not only automatic and computationally tractable, but also effectively to mitigate the interference caused by compilers and their optimization options.
- We supervise representation learning and similarity measurement via SNN, and use the trained networks to infer the similarities of smart contract pairs.
- We formulate our positive and negative sample pairs from smart contract collection using Etherscan, and conduct extensive experimental evaluations on them, which demonstrate the robustness and effectiveness of SmartSD on smart contract similarity detection.

## II. Motivation and Problem Statement

In this paper, we investigate Ethereum smart contracts developed in the high-level language Solidity. Due to the significant differences introduced by different major Solidity versions, the differences existing in smart contract bytecodes compiled by different Solidity compiler versions may be also significant. To understand such differences, we empirically compile a smart contract's source code using two different compiler version with and without optimization options enabled. Accordingly, we observe that the opcodes generated across these compilers are different regarding opcode numbers, opcode types, and the occurring frequencies of the same opcode. Some specific opcodes reside only in the sequence compiled by certain versions (e.g., Solidity compilers earlier than version 0.4 do not produce opcode `revert`). In addition, the opcode combinations and their positions are different across different Solidity compiler versions. These observations imply that direct birthmarks using signatures or fingerprints on bytecodes may suffer from the susceptibility to the interference introduced by opcodes and weak generalizability, and higher-level yet difference-tolerant representations are needed to address this limitation.

Our goal here is to construct the similarity detection model over Ethereum smart contract bytecodes: we leverage deep neural networks of superior feature learning ability to learn the latent representation from the abstracted opcode sequence for each smart contract's bytecodes, and devise SNN to supervise representation learning and similarity measurement. More specifically, given two smart contracts' bytecodes $c_i$ and $c_j$, the opcode sequences $s_i$ and $s_j$ are first extracted and abstracted from $c_i$ and $c_j$ respectively, and then jointly embedded into unified vector space so that we can reasonably measure the similarity between them. Formally, the similarity measurement of $s_i$ and $s_j$ can be formulated as:

$$s_i \xrightarrow{\phi(s_i)} \mathbf{x}_i \rightarrow \mu(\mathbf{x}_i, \mathbf{x}_j) \leftarrow \mathbf{x}_j \xleftarrow{\phi(s_j)} s_j \tag{1}$$

where $\phi : s \rightarrow \mathbf{x} \in \mathbb{R}^d$ is representation learning function to map opcode sequence $s$ into $d$-dimensional vector space, and $\mu(\cdot)$ is similarity measuring function. We exploit SNN to design $\mu(\cdot)$. The similarity detection can thus be stated in the form of $\mu : (\phi(s_i), \phi(s_j)) \rightarrow y$, which outputs the similarity score of the input sample pair in the class space $y \in \{0, 1\}$ representing the different and similar classes respectively. That is, if $s_i$ and $s_j$ are similar, then $\mu(\phi(s_i), \phi(s_j)) \rightarrow 1$; otherwise, $\mu(\phi(s_i), \phi(s_j)) \rightarrow 0$.

## III. Methodology

In this section, we technically detail how we perform representation learning and similarity detection on smart contract bytecodes though SNN in our designed model SmartSD. Fig. 1 specifies the overview of the SmartSD.

### A. Data Preprocessing

In order to prepare the smart contracts for our experimental evaluations on SmartSD, we first collect our dataset using Etherscan, which is a block explorer and analytics platform for Ethereum. More specifically, the HTML pages describing the transaction information of smart contracts are first crawled and parsed, such that the data regarding smart contracts' bytecodes, and their source codes and compiler versions used are then collected. Each contract's bytecodes are encoded by a string of hexadecimal bytes, which do not reflect the underlying operations of Ethereum. Therefore, it is inadvisable to analyze these binaries directly on their raw bytes. In this regard, we disassemble the contract's bytecode into assembly EVM instructions first, and perform the representation learning on them. Currently, the EVM defines about 142 instructions, and there are more than 100 instructions to be extended. For the ground truth preparation, we compile a smart contract source code to generate bytecodes using compilers of different versions, where we consider bytecode pairs from them as positive (similar) data and bytecodes compiled from different source code as negative (different) data. More details are presented in Section IV-A.

### B. Representation Learning

Given the instruction set extracted from the bytecode of each smart contract, it is reasonable to learn the embedding from the sequence as the desired representation for each smart contract. However, such an implementation applied directly on the

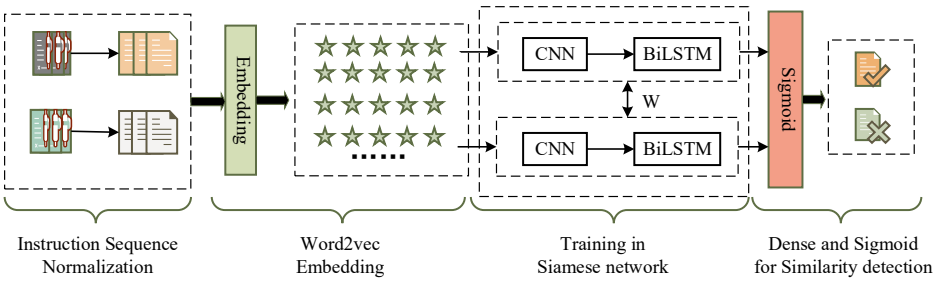


Fig. 1. The overview of SmartSD

instruction set can not expressively capture the correlations and variations among smart contracts. As we want to make features reflect the smart contract semantics instead of functionality, using all instructions as they exactly appear may expose us to the exhaustive functionality information. This immensely increases the representation learning complexity, and decreases the embedding expressiveness, which may degrade the successive SNN performance in turn. To this end, the instruction set of each smart contract is initially abstracted to the sequence of opcodes. Afterwards, the opcode sequence is fed to deep learning framework for representation learning.

*1) Instruction Abstraction:* Each instruction contains an opcode followed by some low-frequency tokens (e.g., numerical constants, memory addresses, and special strings), which can be seen on the left-hand side of Fig. 2. These low-frequency tokens are random and have a very insignificant relationship with the semantics of the program. In this respect, we design the rule to abstract each instruction for the subsequent representation learning as follows: the opcode remains unchanged, and all the non-opcode tokens in the instruction are removed. An example in Fig. 2 illustrates the instruction abstraction processing: the instruction sequence on the left is disassembled from smart contract bytecodes; the abstracted instruction sequence on the right is simply composed of opcodes. For example, using the designed abstraction rule, the instruction "PUSH1 0x80" becomes "PUSH1", and the instruction "PUSH1 0xf" is changed to "PUSH1". After the instruction abstraction, we can represent a smart contract $c$ as an opcode sequence $s = \{op_1, op_2, op_3, \cdots, op_n\}$. In the sequence, each $op_i$ represents the opcode of the $i_{th}$ instruction within $n$ total number of instructions.

*2) Opcode Embedding:* After getting the opcode sequences for smart contracts, we further transform them into numerical embedding space that neural network is able to understand and process. To this end, we first perform embedding operation to map each unique opcode to a vector, such that the opcode sequence can be comprehensively represented. Specifically, SmartSD employs the representative skip-gram model [17] to learn opcode representations to encode their contextual relatedness. Given a set of opcode sequences, each of which is $s = \{op_1, op_2, op_3, \cdots, op_n\}$, we feed them to the skip-gram model, and obtain a $k$-dimensional vector for each unique opcode by evaluating an opcode's neighborhood co-occurrence

| | |
|---|---|
| PUSH1 0x80 | PUSH1 |
| CALLVALUE | CALLVALUE |
| DUP1 | DUP1 |
| ISZERO | ISZERO |
| PUSH1 0xf | PUSH1 |
| ..... | ..... |
| DUP1 | DUP1 |
| REVERT | REVERT |
| PUSH6 0x627a7a723058 | PUSH6 0x627a7a723058 |
| SHA3 | SHA3 |
| PUSH27 | PUSH27 |
| 0x227f8a994da323974f6 | 0x227f8a994da323974f6 |
| ba452a809c3e4b68349ee | ba452a809c3e4b68349ee |
| 3fdd985e5a911e | 3fdd985e5a911e |

Fig. 2. Abstraction for smart contract instructions.

within a window $w$ conditioned on its current embedding. In this way, the learning objective of skip-gram is defined as:

$$\underset{\psi}{\operatorname{argmin}} - \sum_{-w \le j \le w, i \ne j} \log p(op_{i+j} | \psi(op_i)), \qquad (2)$$

where $\psi(op_i)$ is $op_i$'s current embedding. After opcode embedding, the opcode sequence of each smart contract can be converted to an $n \times k$-dimensional vector.

*3) Representation Learning for Opcode Sequences:* Among neural networks, convolutional neural network (CNN) [16] can capture the local correlation, while Long Short-term Memory (LSTM) [9] can encode the sequential dependency, which best fit in our problem. We thus design a model that leverages the advantages of CNN and LSTM over the opcode embedding sequences for smart contract representation learning. The model network structure diagram for representation learning is shown in Fig. 3.

We first enable CNN, which stacks a convolutional layer and a normalization layer, to refine the opcode embedding sequence of each smart contract with locally aggregated opcode information. In this way, it crafts more informative and higher-level embedding space to facilitate the following sequence modeling. To characterize the interactions among different opcode grams, we further integrate filters of different kernel sizes into CNN to enrich the feature semantics for smart contracts. Taking the opcode sequence embedding matrix $\mathbf{S} \in \mathbb{R}^{n \times k}$, the convolutional layer adopts $m$ filters of shape $l \times k$ to perform convolution operations on $\mathbf{S}$, and formulates a new embedding matrix $\mathbf{S}^* \in \mathbb{R}^{(n-l+1) \times m}$ with kernel size $l$.

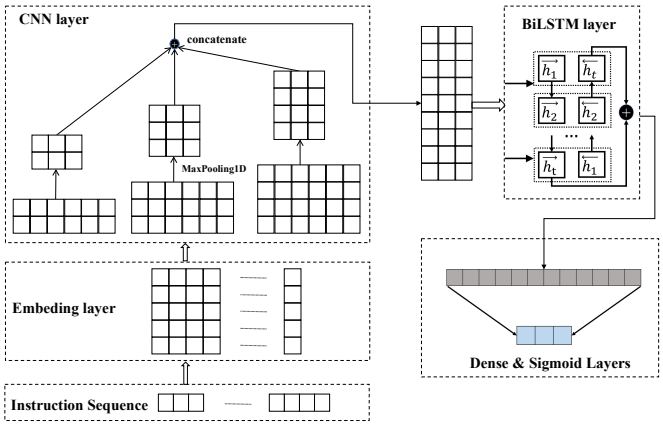


Fig. 3. Representation learning using CNN and BiLSTM

To extract multi-view feature patterns from 2, 3 and 4 opcode grams, we employ kernels of size 2, 3 and 4 to convolute $\mathbf{S}$.

Afterward, the resulting feature matrix $\mathbf{S}^* = [\mathbf{e}_1, \cdots, \mathbf{e}_n]^T$ from CNN is fed to bidirectional LSTM (BiLSTM) to embed the sequential dependency, and output the desired representation. The BiLSTM proceeds by (1) reading $\mathbf{S}^*$ through the composite non-linear transformations $\mathcal{H}$ to learn a hidden vector $\mathbf{h}_t$ at timestep $t$: $\mathbf{h}_t = \mathcal{H}(\mathbf{e}_t, \mathbf{h}_{t-1}), \quad \mathbf{h}_t \in \mathbb{R}^d$ [9]; (2) devising two LSTMs with one processing the sequence in a forward direction and the other in a backward direction to jointly capture bidirectional dependencies and provide additional context to the network; and (3) concatenating the forward and backward hidden vectors at timestep $t$ into new $\mathbf{h}_t = \left[\overrightarrow{\mathbf{h}_t}; \overleftarrow{\mathbf{h}_t}\right]$. As it entirely reads the input sequence in both directions, the hidden states $\mathbf{h}_n$ at the last timestep act as the summary vector to represent the opcode sequence.

### C. Supervised Learning using Siamese Neural Networks

Under the supervise-learning setting, we use siamese neural networks (SNN) [1], [12] to chain and optimize the full learning procedure, including representation learning, similarity measurement, and similarity prediction. There are four reasons behind this network choice. (1) SNN has been successfully deployed in the similarity-based applications, such as zero-shot and few-shot image recognition. (2) SNN supports back-propagation optimization for the aforementioned representation learning process to update its parameters. (3) SNN of twin networks trains the unilateral network by sharing weights [28]; this ensures that smart contract similarity measurement can process two opcode sequences consistently and the weights of both parties are consistent as well. (4) After obtaining the high-level representations of smart contracts, SNN can calculate their similarity in a parameterized manner [22], instead of simply calculating the similarity scores without fine-tuning in an error-prone way.

In our model design, we elaborate a deep SNN, whose identical subnetworks receive two opcode sequence embedding matrices $\mathbf{S}_i$ and $\mathbf{S}_j$ of smart contract bytecodes $c_i$ and $c_j$, and pass them through CNN and BiLSTM in succession to learn the representations $\mathbf{x}_{s_i}$ and $\mathbf{x}_{s_j}$ respectively that extract high-level and difference-tolerant features. Different from the conventional SNN performing direct similarity metric computes [4], our network's top conjoining layer devises a multilayer perceptron (MLP) [8] stacking multiple fully connected layers to fuse features as:

$$\mathbf{x} = \left[\mathbf{x}_{s_i}; \mathbf{x}_{s_j}\right] \tag{3}$$

and measure the similarity using:

$$\mu_\theta\left(\mathbf{x}_{s_i}, \mathbf{x}_{s_j}\right) = \mathrm{MLP}_\theta\left(\mathbf{x}\right) \tag{4}$$

where $\theta$ are parameters introduced by the network. This is followed by a sigmoid activation function mapping onto the interval $[0, 1]$, which is used to minimize cross-entropy loss. Given the training opcode sequence pair from the corresponding smart contracts' bytecodes $(s_i, s_j) \in D$, and the similarity label $y \in Y$ where $y = 0$ means that $s_i$ and $s_j$ are different, and $y = 1$ means that $s_i$ and $s_j$ are similar, the cross entropy loss of our similarity detection (i.e., binary classification problem) can be defined as:

$$\mathcal{L} = -\sum_{(\mathbf{X}_p^I, \mathbf{X}_q^I) \in \mathcal{D}} y \log(P) + (1-y)\log(1-P) \tag{5}$$

$$\text{s.t.} \quad P = \sigma(\mu_\theta(\mathbf{x}_{s_i}, \mathbf{x}_{s_j})) \tag{6}$$

where $\sigma$ is the sigmoid activation function, and the parameters introduced by representation learning and similarity measurement can be comprehensively updated via gradient descent algorithms (e.g., Adam).

From the model formulation, it is worth remarking two significant advantages yielded by our methodology: (1) smart contract representation learning and similarity measurement between sample pairs are automated and advanced by deep learning frameworks without prior domain knowledge; and (2) the task-specific representations learned by the designed networks can tolerate the difference caused by compilers and their optimization options, and generalize well to unseen data..

## IV. Evaluation

### A. Dataset Preparation and Experimental Settings

As a supervised deep learning based scheme is adopted by SmartSD, a large dataset consisting of totally 72,612 samples is thus constructed to boost the training and testing of our method. Specifically, the following steps are enforced in preparing the samples with ground-truth labels:

- To correctly prepare positive (contract pairs that are really similar) and negative (contract pairs that are indeed different) samples, we utilize smartEmbed [6], a tool that detects smart contract clones based on their Solidity source code, to identify similar and dissimilar smart contracts crawled from Etherscan. In our setting, their detected smart contract pairs, which are not identical but with similarity values greater than 0.95, are taken as candidates of positive samples $CP$; while the detected pairs with similarity values smaller than 0.35, are taken as candidates of negative samples $CN$.

- On the basis of the candidate positive and negative samples, we further retrieve their corresponding runtime bytecode from the public Ethereum Cryptocurrency database[2] that is hosted on the Google BigQuery by feeding in their contract addresses. The bytecodes of a candidate smart contract pair is then organized into a triplet of $\langle bin(p), bin(q), l\rangle$, where $l \in \{-1, +1\}$ is the ground truth label that indicates whether the smart contracts $p$ and $q$ belongs to the positive or the negative pair, while $bin(.)$ denotes the runtime bytecode of a smart contract.
- To improve the ability of the trained model in dealing with the adverse impacts from different compiler settings, we further attempt to incorporate positive and negative samples by compiling a smart contract's source code with varying compiler settings. To this end, we randomly pick equivalent number of smart contract pairs from $CP$ and $CN$, and then try to compile them with varying Solidity compiler versions as well as setting the *–optimize* option on and off. Specifically, 5 different Solidity compiler versions, including *0.4.24*, *0.5.6*, *0.6.4*, *0.7.2* and *0.8.2*, are set up to compile the source code of each picked smart contract individually[3]. The successfully compiled ones are then combined to make up positive and negative triplets accordingly.

With these above steps, we finally manage to produce 42,082 pairs of positive samples and 30,530 pairs of negative samples, as our dataset.

For experimental settings, the dataset is randomly divided into training, validation and test set at a ratio of 80%, 10%, and 10%. The model is trained using a RTX3090 GPU with the Adam optimizer. The batch size is set to 64, and the initial learning rate is set to 0.001. In each epoch, we shuffle the training samples while calculate the accuracy on the validation set. Besides, to avoid the over-fitting and non-convergence problems, early stopping is enforced that stops model training right after the epoch that the validation accuracy no longer improves. Finally, the model with the highest accuracy witnessed during all the epochs is adopted, with which frequently used performance metrics including accuracy, precision, recall, and F1 score are evaluated and reported on the test set. Also, the opcode embedding model is trained using 100 epochs and 6 context window size.

### B. Evaluation Results

In this section, we report the performance of SmartSD. In addition, we compare the performance of SmartSD with variant models by simply substituting the CNN+BiLSTM structure in our SmartSD with other widely used deep neural network structures, including the pure CNN, BiLSTM, GRU and Bi-GRU. We denote them as SmartSD$_{CNN}$, SmartSD$_{BiLSTM}$, SmartSD$_{GRU}$ and SmartSD$_{BiGRU}$, respectively.

TABLE I
PERFORMANCE COMPARISON WITH VARYING NEURAL NETWORK STRUCTURES IN SMARTSD

| *Model* | *Accuracy* | *Precision* | *Recall* | *F1-score* |
|---|---|---|---|---|
| *SmartSD* | **98.37%** | **0.9889** | **0.9813** | **0.9850** |
| *SmartSD*$_{CNN}$ | 93.09% | 0.9321 | 0.9119 | 0.9219 |
| *SmartSD*$_{BiLSTM}$ | 95.20% | 0.9536 | 0.9383 | 0.9459 |
| *SmartSD*$_{GRU}$ | 93.14% | 0.9352 | 0.9071 | 0.9209 |
| *SmartSD*$_{BiGRU}$ | 94.32% | 0.9479 | 0.9337 | 0.9407 |

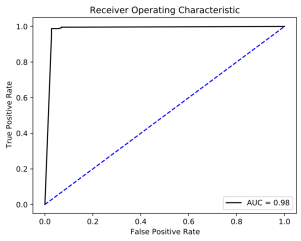

Fig. 4. Performance regarding ROC

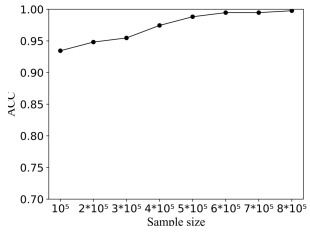

Fig. 5. Training size analysis

Table I summaries the experimental results. As the data show, SmartSD as well as its substituted models all exhibit rather good detection performance with respect to the evaluation metrics mentioned in Section IV-A. Their accuracy values generally compete with or outperform the 93.27% accuracy value as reported by EClone [14], [15], a smart contract similarity detection method that adopts inefficient and sophisticated symbolic execution techniques. This indicates the potency of adopting a deep learning based way to achieve smart contract similarity detection task. Especially, the original SmartSD that adopts a CNN+LSTM structure for encoding the opcode sequence of a smart contract's bytecodes, outperforms all the other substitute models with a relatively obvious margins. Its 98.37% high detection accuracy and 0.9850 F1-score value indicate the superiority of combing CNN and LSTM structure to capture compiler-setting-agnostic features, which makes our method highly resilient against the disturbance of varying compiler settings. Additionally, as depicted in Fig. 4 for the ROC curve, SmartSD achieves a very high AUC (Area Under the Curve) value of 0.98.

The impact of the training set size on the detection performance of SmartSD is also evaluated, by training SmartSD with gradually increased number of training samples and observing corresponding detection accuracy on the same test set. As depicted in Figure 5, as the number of training samples increases, the detection performance of SmartSD increases. It indicates the importance of abundant data for deep learning based methods, while the availability of the massive diversiform open-sourced smart contracts on Etherscan makes deep learning especially suitable for our problem.

[2]https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics

[3]As pointed out in a recent study [21], there are many minor Solidity compiler versions within each major version (For example, besides 0.4.24, there are 26 more officially released minor versions in the major version 0.4.x), but their impacts on the compiled bytecodes are insignificant, our used minor versions are thus randomly selected as long as there is one for each major version. Also, note that not all compiler versions can always successfully compile each smart contract, especially earlier versions are not used considering their immaturity and the high failure rate in compiling the smart contract in our dataset.

## V. Related Work

**Similarity analysis of smart contracts:** Code similarity analysis has always been a long-term research topic, but there have been few studies [6], [14], [15] conducted on the emerging smart contracts. Liu et al. [14], [15] defined the concept of smart contract birthmarks by borrowing the typical definition of software birthmark. They proposed a symbolic transaction sketch technique to achieve smart contract similarity detection and DApp (Decentralized Application) clone detection on the bytecode level. Different from their methods that resorts to symbolic execution and expert domain knowledge, SmartSD recurs to the deep neural networks' powerful learning ability and the availability of many smart contract to achieve high accurate and efficient similarity detection while with less human intervention. Gao et al. [6] proposed to use unsupervised embedding algorithms including word2vec and Fasttext to encode smart contracts into numerical vectors, on the basis of which similarity between smart contract pairs can be efficiently computed with an Edulidean distance based similarity metric. Different from their works that can only operates on the Solidity source code, we adopt a deep siamese neural network architecture that works on the actually deployed smart contracts' bytecodes, with the aim of defeating the disturbance from varying compiler settings.

**Smart contract bug detection**: With the developments of smart contracts and the widely used yet maturing programming languages, their defects and security issues have attracted major research attentions for the solutions. Apart from these conventional methods [13], [18], [23], [24] that generally detect bugs or vulnerabilities based on symbolic execution, formal verification, and manually constructed bug patterns or specifications, Gao et al. [7] attempts to detect/retrieve bugs from smart contracts' source code by checking the similarity of the smart contract against known buggy contracts. SmartSD can accurately detect the similarity of smart contracts directly on the deployed bytecode, thus it is promising to be applied to achieve similarity checking based known bug search in the scenario that smart contracts' source codes are unavailable.

## VI. Conclusion

In this paper, we propose to detect the similarity of Ethereum smart contracts and build up a bytecode-level model SmartSD using deep siamese neural network to supervise the representation learning and the similarity measurement process. The experimental results show that SmartSD achieves 98.37% high detection accuracy and 0.9850 F1-score, which demonstrate its effectiveness of smart contract similarity detection; SmartSD also significantly outperforms the baseline models, and is computationally tractable and effectively mitigates the interference caused by compilers.

## Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (61702414), the Science and Technology of Xi'an (2019218114GXRC017CG018-GXYD17.16), the Natural Science Basic Research Program of Shaanxi (2022JM-342, 2018JQ6078, 2020JM-582), the International Science and Technology Cooperation Program of Shaanxi (2019KW-008), the Key Research and Development Program of Shaanxi (2019ZDLGY07-08), and Special Funds for Construction of Key Disciplines in Universities in Shaanxi.

## References

[1] J. Bromley, J. W. Bentz, L. Bottou, I. Guyon, Y. LeCun, C. Moore, E. Säckinger, and R. Shah, "Signature verification using a "siamese" time delay neural network," *IJPRAI*, vol. 7, no. 04, pp. 669–688, 1993.

[2] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *TSE*, 2020.

[3] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *SANER*. IEEE, 2017, pp. 442–446.

[4] S. Chopra, R. Hadsell, and Y. LeCun, "Learning a similarity metric discriminatively, with application to face verification," in *CVPR*, 2005.

[5] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *ICSE*, 2020, pp. 530–541.

[6] Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding," in *ICSME*, 2019.

[7] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *TSE*, pp. 1–18, 2020.

[8] V. Garcia and J. Bruna, "Few-shot learning with graph neural networks," *arXiv preprint arXiv:1711.04043*, 2017.

[9] A. Graves, "Generating sequences with recurrent neural networks," *arXiv preprint arXiv:1308.0850*, 2013.

[10] N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, "Characterizing code clones in the ethereum smart contract ecosystem," in *FC*, 2020.

[11] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." in *Ndss*, 2018, pp. 1–12.

[12] G. Koch, R. Zemel, R. Salakhutdinov *et al.*, "Siamese neural networks for one-shot image recognition," vol. 2. Lille, 2015.

[13] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *USENIX Security*, 2018, pp. 1317–1333.

[14] H. Liu, Z. Yang, Y. Jiang, W. Zhao, and J. Sun, "Enabling clone detection for ethereum via smart contract birthmarks," in *ICPC*, 2019.

[15] H. Liu, Z. Yang, C. Liu, Y. Jiang, W. Zhao, and J. Sun, "Eclone: Detect semantic clones in ethereum via symbolic transaction sketch," in *ESEC/FSE*, 2018.

[16] Y. Luan and S. Lin, "Research on text classification based on cnn and lstm," in *ICAICA*. IEEE, 2019, pp. 352–355.

[17] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv:1301.3781*, 2013.

[18] 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 *ASE*, 2019.

[19] K. Sliwak, "Smart contract reentrancy: Thedao," https://medium.com/@zhongqiangc/smart-contract-reentrancy-thedao-f2da1d25, 2021.

[20] Solidity, "Github - ethereum/solidity: Solidity, the contract-oriented programming language," https://github.com/ethereum/solidity, 2021.

[21] Z. Tian, J. Tian, Z. Wang, Y. Chen, H. Xia, and L. Chen, "Landscape estimation of solidity version usage on ethereum via version identification," *IJIS*, 2021.

[22] Z. Tian, Q. Wang, C. Gao, L. Chen, and D. Wu, "Plagiarism detection of multi-threaded programs via siamese neural networks," *IEEE Access*, vol. 8, pp. 160 802–160 814, 2020.

[23] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: static analysis of ethereum smart contracts," in *WETSEB*, 2018, pp. 9–16.

[24] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, and et al., "Securify: practical security analysis of smart contracts," in *CCS*, 2018.

[25] M. Wohrer and U. Zdun, "Smart contracts: security patterns in the ethereum ecosystem and solidity," in *IWBOSE*, 2018, pp. 2–8.

[26] G. WOOD, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, pp. 1–32, 2014.

[27] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, pp. 1–32, 2014.

[28] L. Zhang, Z. Feng, W. Ren, and H. Luo, "Siamese-based bilstm network for scratch source code similarity measuring," in *IWCMC*. IEEE, 2020, pp. 1800–1805.