DeeSCVHunter: A Deep Learning-Based Framework for Smart Contract Vulnerability Detection

Xingxin Yu[1,2], Haoyue Zhao[1,2], Botao Hou[1,2], Zonghao Ying[1,2] and Bin Wu[1,2](✉)
[1] *State Key Laboratory of Information Security,*
Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] *School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China*
{yuxingxin,zhaohaoyue,houbotao,yingzonghao,wubin}@iie.ac.cn

2021 International Joint Conference on Neural Networks (IJCNN) | 978-1-6654-3900-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/IJCNN52387.2021.9534324

***Abstract*—Security attacks in smart contracts have drawn extensive attention due to the financial loss and erosion of trust caused by vulnerabilities. Even worse, smart contract is a tamper proof digital agreement and fixing bugs in it is difficult, so it is necessary for developers to detect security vulnerabilities in smart contract before deployment. Researchers have proposed several methods on smart contract vulnerability detection. However, despite the numerous vulnerability-finding tools, few of them have ideal detection performance because most of them rely on fixed rules, which is inefficient. In this paper, a modularized and systematic Deep Learning-based framework is proposed to automatically detect smart contracts vulnerability, called DeeSCVHunter. Particularly, we focus on two types of smart contract vulnerabilities: reentrancy and time dependence. And we propose a novel notion of Vulnerability Candidate Slice (VCS) to help models capture the key point of vulnerability. We conduct experiments on real-world dataset and the results prove the effectiveness of VCS, which leads to 25.76% improvement in F1-score at most. And extensive experiments also show that our proposed framework significantly outperforms state-of-the-art methods.**

I. INTRODUCTION

The past few years, blockchain technology has developed rapidly due to its decentralization and tamper-free nature [1]. Blockchain is essentially a distributed and shared transaction ledger, maintained by all the miners in the blockchain network following a consensus protocol [2]. Recently, people have witnessed a dramatic rise in the popularity of smart contracts [3]. Smart contract is a tamper proof digital agreement which is also a program automatically running on the blockchain. And Ethereum [4] is the most popular framework for executing smart contracts.

Unfortunately, similar to all software, smart contracts may contain bugs. And ill-designed smart contracts will expose vulnerabilities, which can be exploited by malicious attackers for financial gains. Even worse, it is nearly impossible to update a smart contract after deploying it on blockchain and transactions on Ethereum are immutable, which means that losses cannot be recovered. There have been many vulnerabilities in smart contracts that have been maliciously exploited in the recent past years. One notable example is the DAO (Decentralized Autonomous Organization) event [5] where the hackers exploit the reentrancy vulnerability of the DAO contract to steal 3.6 million Ether (Cryptocurrency of Ethereum). According to SlowMist Hacked [6], blockchain networks have suffered more than 10 billion USD losses due to the security vulnerabilities. Therefore, there is a compelling need to discover and detect smart contract vulnerabilities in advance.

Many approaches and works have been proposed to find vulnerabilities in smart contracts [7]-[11]. These methods are mainly inspired by existing testing methods from the programming language community, such as symbolic execution [8][11][17] and dynamic execution [12][13]. However, despite the prevalence of these analysis tools, vulnerabilities abound in smart contracts [14]. We investigate these methods and observe that most of them mainly rely on expert knowledge and manual experience to extract fixed rules of vulnerabilities, which is inefficient and laborious. What's more, using expert-defined hard rules crudely tends to introduce false-positives and false-negatives. Additionally, whether the defined rules are representative requires experts' judgment, which is difficult to draw analogy in batches.

A method that is good at detecting contract vulnerabilities can make the contracts published on blockchain more robust and secure. However, detecting vulnerabilities of smart contract is still a huge challenge, which motivates this paper. Aiming at solving above limitations, we propose a fully automatic deep learning based vulnerability detection framework, called DeeSCVHunter. We capsule different types of deep learning (DL) models in our framework, including convolutional neural network and recurrent neural network, etc., to learn the patterns of vulnerable smart contract samples without experts and external knowledge. And we propose a notion of vulnerability candidate slice (VCS), which leverages data dependencies and/or control dependencies in source code to improve the performances of DL models. We conducted experiments on a public available real-world dataset with code and label pairs [15], which contains two types of vulnerabilities (i.e., reentrancy and time dependence). The result shows that our framework significantly outperform state-of-the-art (SOTA) methods on the detection of both two types of vulnerabilities.

The main contributions of this paper are summarized as follows:

✉ Corresponding author.
E-mail address: wubin@iie.ac.cn.

978-0-7381-3366-9/21/$31.00 ©2021 IEEE

Authorized licensed use limited to: Japan Advanced Institute of Science and Tech. Downloaded on July 27,2024 at 08:37:48 UTC from IEEE Xplore. Restrictions apply.

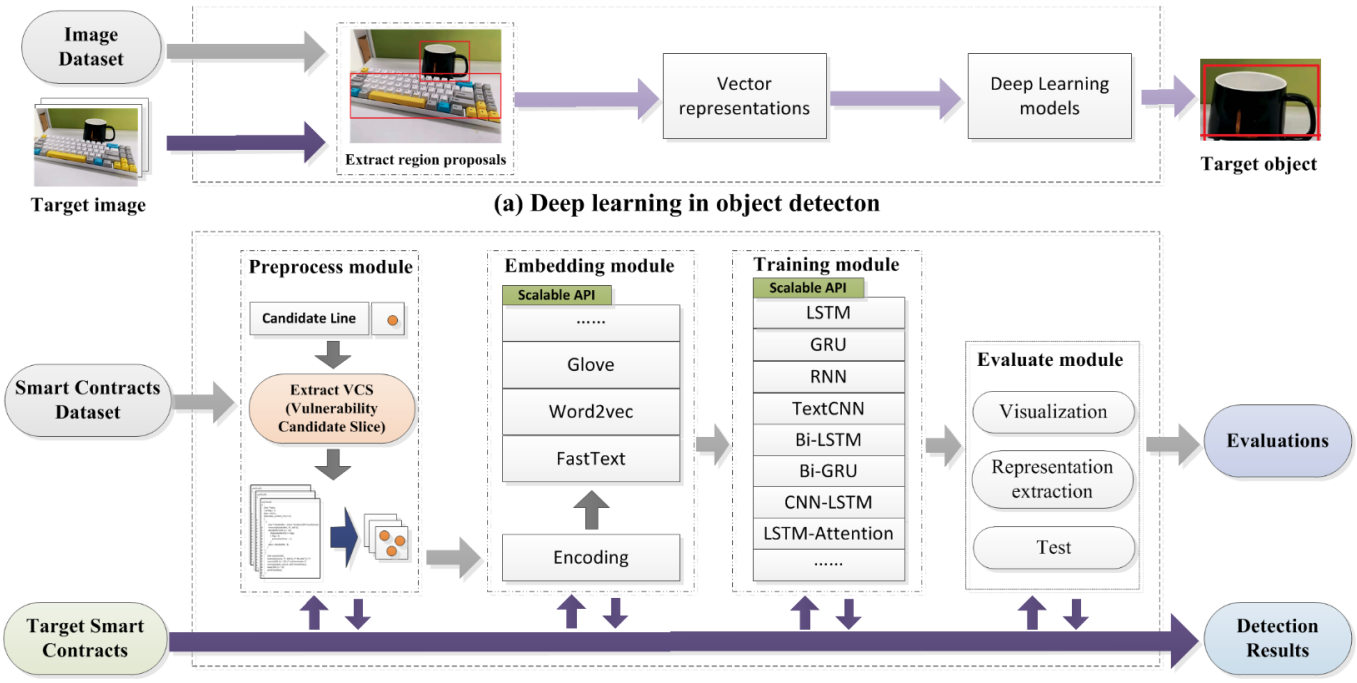

Fig. 1: (a) The process of object detection and the region proposals. (b) The DeeSCVHunter framework inspired by the notion of region proposal and extract vulnerability candidate slice by the data flow and/or control flow in Solidity source code of smart contract.

- To the best of our knowledge, we develop the first systematic and modularized framework for detecting smart contract vulnerabilities based on DL, called DeeSCVHunter, which capsules eight mainstream DL models and three embedding schemes.
- We propose a notion of vulnerability candidate slice (VCS), which is a new feature for recognizing vulnerable contracts. VCS contains richer syntax characteristics and semantic information that can significantly improve the performance of DL models in smart contract vulnerability detection.
- We compare our framework with other seven state-of-the-art methods and the experiment results show that our framework performs best in the detection of two types (reentrancy and time dependence) of smart contract vulnerabilities.
- We have made the code of DeeSCVHunter and the dataset used in our experiments publicly available : https://github.com/MRdoulestar/DeeSCVHunter.

This paper is organized as follows. Section II introduces the related works. Section III introduces the framework architecture, Section IV introduces the details of our method. Section V describes the experimental results and discussions. Finally, we conclude and summarize the paper in Section VI.

II. Related Work

Currently, smart contract has become a research hotspot and smart contract vulnerability detection is one of the most important problems in block chain. Up to now, many works have been proposed to solve the security issues of smart contract. Lu et al. [8] analyzed four kinds of smart contract vulnerabilities and proposes the first tool based on symbol execution, called Oyente. Oyente performs symbolic execution on contract functions and marks errors based on simple patterns. Additionally, Securify [11], Maian [16], Mythril [35] and Osiris [17] are symbolic execution methods as well. However, above-mentioned methods suffer from high false negative rates due to the inability to explore all possible program paths. And Securify also requires manually defined patterns to search vulnerabilities, which brings high false negative rates.

Tikhomirov [10] proposed Smartcheck, which is an extensible static analysis tool that has competence to detect more than 20 types of vulnerabilities. Smartcheck first translates source code of smart contract written in Solidity into an XML-based intermediate representation. Then Smartcheck uses the representation checking it with XPath patterns to determine if it has vulnerabilities. Feist et al. [7] proposed a static analysis framework, called Slither. Slither will parse the code to get the Abstract Syntax Tree of the Solidity, inheritance graph, Control Flow Graph and function list, then the code will be converted to intermediary language to realize high-precision analysis through a simple API. Then Slither analyzes the relationship between variables and functions through predefined rules, and obtains the results.

Recent works also explore dynamic execution for vulnerability detection, such as [12][13]. But these methods require a hand-crafted agent contract for reentrancy detection, preventing it from fully automated application. Samreen et al. [18] proposed a tool that combines static and dynamic analysis to detect the reentrancy vulnerability. It first uses TXL (Turing eXtender Language) to analyze smart contracts to identify

Authorized licensed use limited to: Japan Advanced Institute of Science and Tech. Downloaded on July 27,2024 at 08:37:48 UTC from IEEE Xplore. Restrictions apply.

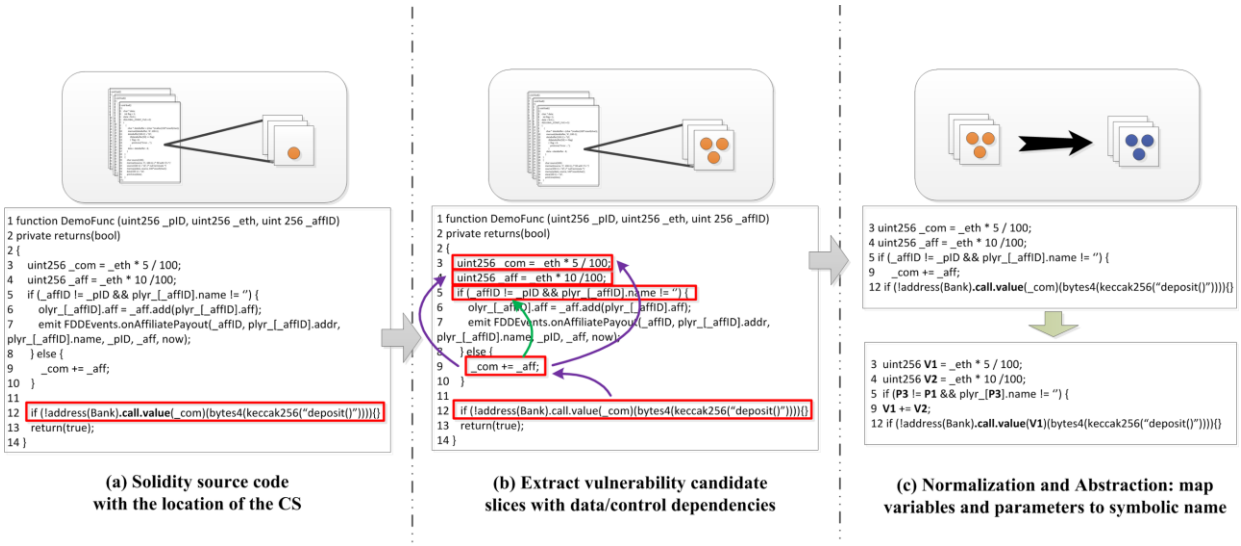

Fig. 2: An example illustrating how we take advantage of candidate statement (a) to extract vulnerability candidate slice (b) by data dependencies and/or control dependencies. We also show the normalization and abstraction process in (c).

potential vulnerable functions, and then automatically generates attack contracts based on the ABI (Application Binary Interface) specifications of the tested smart contracts. Finally it applies dynamic analysis to accurately identify reentrant vulnerabilities by interacting with the original contracts.

With the develop of deep learning, a few of researchers have applied it into the detection of smart contract vulnerability. Huang et al. [19] converted the bytecode of vulnerability contract into RGB (red, green, blue) images. And these images are trained by convolutional neural network algorithm to obtain a vulnerability detection model. What's more, Tann et al. [20] applied Long Short Term Memory (LSTM) [24] model to learn vulnerabilities in sequence, which has 97% detection accuracy. However, these methods do not consider the impact of local code vulnerability on the overall code, which makes these methods less interpretable. Zhuang et al. [15] explored using graph neural networks for smart contract vulnerability detection. They constructed a contract graph to represent both syntactic and semantic structures of a smart contract function, and proposed a temporal message propagation network (TMP) to detect vulnerability. However, the process to build contract graph is complicated.

Although the above works have been proposed to detect smart contract vulnerabilities, most of them still rely on expert knowledge and rules. In contrast, deep learning is good at learning latent patterns, which is promising in detecting smart contract vulnerabilities automatically. However, up to now, few of works apply deep learning methods in this task and lack of systematic works. In this paper, we propose DeeSCVHunter to fill this gap.

III. The Framework

In this section, we will introduce the design of the proposed framework and our proposed Vulnerability Candidate Slice (VCS) extracting approach, which can be utilized for significantly improving the performance of smart contract vulnerability detection.

A. Overview

Fig. 1(b) shows the modularized implementation of DeeSCVHunter. Our framework contains four modules: the preprocess module, the embedding module, the training module and the evaluate module. The first module includes the VCS of smart contract extracting approach. The second module is a scalable module and it includes different kinds of word embedding models used for code embedding. The third module is also scalable and it achieves model training. The fourth module aims at visualizing and analyzing detection results.

B. System Flow

In the domain of image processing, we can extract region proposals from image, and regard each region proposal as minimum granularity to train DL models for objects detection as shown in Fig. 1. Inspired by the notion of region proposal in object detection task, we innovatively divide a smart contract into smaller pieces of code (i.e., statements) to generate vulnerability candidate slice (VCS), which like what region proposal is to images, at the preprocess stage in our framework. The VCS is extracted by control dependencies and data-dependencies, which makes VCS contain more syntax and semantics characteristics of vulnerabilities. Particularly, we define transaction amount related constant *call.value* and blockchain timestamp related constant *block.timestamp* as the match pattern for reentrancy vulnerability and time dependence vulnerability respectively. Then, based on the matched statements, we obtain other statements related to this statement by tracing data flows and/or control flows. All these statements build up our VCS.

At the embedding stage, it is a common practice to change raw inputs, such as code tokens, into vector representations, which are acceptable by the deep learning (DL) models. Also, the embedding module in our framework is built to convert code tokens to vectors. More importantly, the vector representations can preserve the code semantics and carry more meaningful information than raw input code sequences. Usually, the semantically similar code tokens will be in close proximity in

Authorized licensed use limited to: Japan Advanced Institute of Science and Tech. Downloaded on July 27,2024 at 08:37:48 UTC from IEEE Xplore. Restrictions apply.

that vector space. The embedding module allow the DL models to learn from a richer source. In this module, we encapsulates three popular word embedding models, including Word2vec model [21], FastText model [22] and GloVe model [23].

At the training stage, we implement eight different kinds of DL models in the training module, which allows users to pick out any of the built-in models for building a vulnerability detector. The eight models including recurrent neural network (RNN) and its four variants (i.e., the Gated Recurrent Unit (GRU) [24], the LSTM [25], the bidirectional GRU (Bi-GRU) and the bidirectional LSTM (Bi-LSTM)), the TextCNN [26], the CNN-LSTM [27] and the LSTM with Attention mechanism [28]. It should be noted that the embedding and training modules in our framework provide easy-to-use APIs, which makes it easily to integrate new embedding schemes and implement new DL models for training. It should be noted that any neural network models implemented using Keras [29] or TensorFlow [30] can be invoked by our framework.

During the evaluation phase, users can convert test data into vector representation by feeding them to preprocess and embedding module firstly, and using a trained model to obtain detection results. The provability of each test smart contract containing the vulnerable code, and the vulnerable function name in the smart contract will be recorded. Our evaluation module also provides APIs for visualizing and analyzing training/validation processes, which is supported by Keras APIs and TensorBoard [31].

IV. Method Implementation

A. Definition

In this paper, we focus on detecting vulnerabilities in smart contract. In order to help understand the details of our framework, we define some definitions as follows.

Definition 1 (Smart Contract, function, statement, token): A smart contract C contains a set of functions $\{f_1,\ldots,f_n\}$, denoted by $C=\{f_1,\ldots,f_n\}$. A function fi, where $1\le i\le n$, is an ordered set of statements $\{s_{i,1},\ldots,s_{i,mi}\}$, denoted by $fi=\{s_1,\ldots,s_{i,mi}\}$. A statement $s_{i,j}$, where $1\le i\le n$ and $1\le j\le m_i$, is an ordered set of tokens $\{t_{i,j,1},\ldots,t_{i,j,wi,j}\}$, denoted by $s_{ij}=\{ t_{i,j,1},\ldots, t_{i,j,wi,j} \}$. Tokens consist of identifiers, operators, constants and keywords.

Definition 2 (Reentrancy vulnerability, Timestamp dependence vulnerability): Reentrancy vulnerability is a well-known vulnerability that caused the infamous DAO attack. In Ethereum, when a smart contract function f_1 transfers money to a recipient contract C_1, the fallback function f_2 of C_1 will be automatically triggered. C_1 may invoke back to f_1 in its fallback function f_2 to reenter f_1 for attacking. Since the current execution of f_1 waits for the transfer to finish, C_1 can make use of the intermediate state f_1 to succeed in attacking.

Timestamp dependence vulnerability exists when a smart contract uses the block timestamp as a triggering condition to execute some critical operations, while the miner in Ethereum has the freedom to set the timestamp of a block within a short time interval (< 900 seconds) [12]. Therefore, malicious miners may manipulate the block timestamps to gain illegal benefits.

Definition 3 (Candidate Statement): Giving a function *f* in smart contract C, if any of the statement *s* in *f* matches predefined vulnerability characteristics (e.g., *call.value* in reentrancy vulnerability and *block.timestamp* in time dependence vulnerability), the matched statement *s* in *f* is candidate statement (CS).

Definition 4 (Vulnerability Candidate Slice): Giving a function *f* with CS in smart contract C, vulnerability candidate slice (VCS) is a set of statements, which are semantically with CS in terms of control dependencies and data dependencies, and VCS $\in f$.

B. Extracting Vulnerability Candidate Slice as New Feature

In the process of reviewing the source code of smart contract, we concluded that some vulnerabilities appear in one function and only a part of statements are related to vulnerabilities in a function. Obviously, these corresponding statements are key points for detecting vulnerabilities. In this paper, we propose VCS to represent these statements to help DL models perform better. In order to help understand the details of our VCS extracting approach, we use Fig. 2 to highlight the process.

Algorithm 1 Extract VCS from function

Input: Function *f* and the CS match pattern *p*
Output: Vulnerability candidate slice *vcs* of function *f*

```
1.  cs ← 0
2.  vcs ← empty set
3.  for each statement s in f do
4.     if p match s then
5.        cs ← s
6.        vcs.append(cs)
7.        break
8.     end if
9.  end for
10. for each statement s in f do
11.    if s has data-dependence with cs then
12.       vcs.append(s)
13.    end if
14.    if s has control-dependence with cs then
15.       vcs.append(s)
16.    end if
17. end for
18. Output vcs
```

Step 1: The first step is to define CS match pattern, while noting that the current version of our framework aims at detecting reentrancy and time dependence vulnerabilities. In specifically, we define *call.value* as the CS match pattern for reentrancy vulnerability and define *block.timestamp* as CS match pattern for time dependence vulnerability. It is easily to modify these patterns and even add new patterns for detecting new kinds of vulnerabilities in our framework.

Step 2: Given the source code of Solidity of smart contract and predefined CS match patterns, we first analyze the source code and parse solidity code by lexical analysis secondly. To achieve it, we build a grammar rule and use antlr4 [34] to robustly parse Solidity source code and extract abstract syntax tree (AST), functions, arguments and user-defined variables. And then for each function, we apply CS match patterns to search statements in these functions to build CS, as shown in Fig. 2(a).

Authorized licensed use limited to: Japan Advanced Institute of Science and Tech. Downloaded on July 27,2024 at 08:37:48 UTC from IEEE Xplore. Restrictions apply.

TABLE I. THE EVALUATION RESULTS OF EIGHT MODELS IN DEESCVHUNTER.

Models	Reentrancy Vulnerability				Timestamp Dependence Vulnerability			
	A (%)	P (%)	R (%)	F1 (%)	A (%)	P (%)	R (%)	F1 (%)
RNN	74.42	58.33	53.85	56.00	75.18	80.05	71.14	74.44
RNN+VCS	81.40	69.23	68.54	69.08	78.78	84.61	75.77	78.66
GRU	67.44	52.64	45.87	46.82	73.02	76.58	68.50	71.86
GRU+VCS	83.72	70.23	75.19	72.58	79.14	84.45	74.23	78.33
LSTM	79.07	75.00	46.15	57.14	69.06	79.57	55.66	64.42
LSTM+VCS	88.37	78.57	84.62	81.48	78.42	80.99	77.32	78.48
LSTM-Attention	81.49	93.60	42.88	56.40	75.18	78.10	73.40	74.87
LSTM-Attention+VCS	86.05	75.19	75.13	75.66	**80.50**	**85.53**	**74.86**	**79.93**
Bi-GRU	76.74	62.79	43.99	51.69	73.02	72.79	75.21	73.42
Bi-GRU+VCS	79.07	68.80	52.26	58.88	75.54	80.29	71.73	75.07
Bi-LSTM	81.40	66.17	75.19	70.23	75.90	83.04	66.91	73.00
Bi-LSTM+VCS	**93.02**	**90.70**	**83.46**	**86.87**	79.86	88.09	70.76	78.32
TextCNN	81.40	85.12	45.87	59.58	69.42	65.13	84.54	73.04
TextCNN+VSC	86.05	88.89	61.54	72.73	71.22	65.12	94.46	76.35
CNN-LSTM	76.74	68.11	45.87	54.26	70.86	71.46	70.94	70.14
CN-LSTM+VCS	81.40	69.23	69.65	69.47	71.22	70.62	77.47	73.49

Step 3: As Fig. 2(b) shows, we aim at extracting statements related to the vulnerability as much as possible, which can be built by the means of data flow and/or control flow analysis of program. These kinds of analysis algorithms [32][33] are well known in C/C++ and we adopted them into Solidity to get VCS as shown in Fig. 2(b), supported by antlr4 [34]. Obviously, VCS extends CS to include statements which are semantically related to the CS, where more semantic information is induced by control dependencies and/or data dependencies. After this stage of processing, we will obtain VCS, which is the subset of function.

C. Abstraction and Normalization

It is worth mentioning that user-defined parameters/variables are unpredictable, which will introduce new word out of vocabulary, leading to more false negatives. It is necessary to perform abstraction and normalization to each VCS before embedding the statements to vector representation. This procedure is shown as Algorithm 1. We identify parameters, user-defined variables from the AST of each function and replace every occurrence of parameters, user-defined variables in VCS with symbols such as *Pn*, *Vn* respectively, where the *n* represents the order of parameter/variable definitions. After abstraction, all the symbols are more regular, which avoids the influence of noises.

Algorithm 2 Abstraction and Normalization

Input: Vulnerability candidate slice *vcs* of a function *f*
Output: Normalized vulnerability candidate slice *vcs*'

```
1.  vcs' ← empty set
2.  for each statement s in vcs do
3.      Remove non-ASCII characters, tabs, comments in s
4.      for each token t in s do
5.          if t is variable or parameter then
6.              t ← Map t to symbolic name
7.          end if
8.      end for
9.      vcs'.append(s)
10. end for
11. Output vcs'
```

After abstraction, the non-ASCII characters, tabs and comments in statements will be removed to prevent the introduction of information unrelated to the vulnerability. Then, our approach becomes more robust and the generated VCS can be used as raw input for downstream word embedding and model training tasks.

D. Embedding and Training

After preprocessing the Solidity source code of smart contracts, the extracted VCS should be represented in vector representations that are acceptable for the input to DL models. In order to convert statements in VCS to vectors, we complete scalable embedding module in our framework and capsule three kinds of embedding models, including FastText model, Word2vec model and Glove model.

Given the normalized and abstracted VCS in Fig. 2(c), we parse each statement in VCS and split it into a set of tokens firstly. Then we concatenate these sets of tokens in the order in which the statements appear. Based on these tokens, we apply embedding models to convert source code into vector representations. It should be noted that we choose FastText [22] model as the default model to embed tokens, which is able to solve out-of-vocabulary (OOV) problem and improve the robustness of vulnerability detection. In this paper, we build eight models in our scalable training module, which covers CNN and RNN as well as Attention. These models take vectors as input and the training process for learning is standard. Finally, we obtain eight trained DL models as smart contract vulnerability detectors. And we will do further evaluation based on them.

E. Testing

The framework proposed in this paper is designed for developers to discover their smart contract source code conveniently before deployment. Given target smart contract source code, the source code will be handled with aforementioned steps (i.e., extracting VCS,

Authorized licensed use limited to: Japan Advanced Institute of Science and Tech. Downloaded on July 27,2024 at 08:37:48 UTC from IEEE Xplore. Restrictions apply.

TABLE II. THE COMPARISON RESULT WITH SEVEN EXISTING STATE-OF-THE-ART METHODS.

DeeSCVHunter vs. SOTA Methods	Reentrancy Vulnerability				Timestamp Dependence Vulnerability			
	A (%)	P (%)	R (%)	F1 (%)	A (%)	P (%)	R (%)	F1 (%)
Smartcheck	66.15	34.15	26.42	29.79	61.08	50.00	29.17	36.84
Oyenete	74.05	55.31	49.06	52.00	60.54	48.94	31.94	38.66
Mythril	63.24	33.33	28.30	30.61	**64.87**	**58.54**	**34.53**	**42.48**
Securify	**76.21**	**60.00**	**50.94**	**55.10**	Null	Null	Null	Null
Slither	58.92	36.47	58.50	44.93	63.55	51.32	33.87	41.20
Osiris	61.62	37.14	49.05	42.27	65.66	49.32	31.54	39.21
TMP [15]	84.21	75.12	83.33	78.84	83.42	75.09	83.42	79.08
Our Bi-LSTM	**93.02**	**90.70**	**83.46**	**86.87**	79.86	88.09	70.76	78.32
Our LSTM-Attention	85.92	75.19	75.13	75.35	**80.50**	**85.53**	**74.86**	**79.93**

abstraction/normalization and embedding). Then, the trained models in our framework will take these embedded vectors as input and output detection results (i.e., "1" as vulnerable and "0" as non-vulnerable).

V. EXPERIMENTS AND RESULTS

Our experiments are designed to answer the following three Research Questions (RQs):

- RQ1: Can our DL-based framework detect multiple kinds of smart-contract vulnerabilities, and how about the performance of different DL models?

  For answering this question, we will use eight different DL models to conduct experiments on two types of vulnerability data sets (i.e., reentrancy and time dependence).

- RQ2: Can the VCS (Vulnerability Candidate Slice) extracted from function make our framework more effective, and by how much?

  For answering this question, we will compare the performance of using the whole function and the performance of using VCS only, which is extracted by control dependencies and data dependencies.

- RQ3: How effective is our framework when compared with the state-of-the-art methods?

  For answering this question, we will compare our framework with existing state-of-the-art smart-contract vulnerability detection methods (i.e., Smartcheck [10], Oyenete [8], Mythril [35], Security [11], Slither [7], Osiris [17] and TMP [15].

A. Experimental Datasets

We conduct experiments on smart contract that have Solidity source code on Ethereum, and the dataset is the same as [15], which is available publicly and denoted as ESC (Ethereum Smart Contracts). ESC contains two types of vulnerabilities (i.e., reentrancy and time dependence). It has 40932 Ethereum smart contracts and including 307, 396 functions. Among the functions, nearly 5, 013 functions with *call.value* (i.e., the CS match pattern of reentrancy vulnerability) in statements and around 4, 833 functions with *block.timestamp* (i.e., the CS match pattern of time dependence vulnerability) in statements. And this dataset not only contains Solidity source, but also the ground truth labels. In our experiments, we partition the data samples into two parts: 80% training data and 20% test data.

B. Experimental Settings

In our framework, we build a training module to compare and evaluate eight DL models with each other in detecting smart contract vulnerabilities, including RNN model, GRU model, LSTM model, Bi-GRU model, Bi-LSTM model, TextCNN model, CNN-LSTM model and LSTM-Attention model. Also, we compare the best DL model in our framework with seven existing smart-contract vulnerability detection methods.

As for evaluation metrics, we use metrics which are widely used in machine learning tasks, including accuracy (A), precision (P), recall (R) and F1 score (F1) to measure the performance of vulnerability detection. And we use Keras with Tensorflow backend to implement our framework. And our experiments are run on a server with Ubuntu 18.04.2 LTS: 1 Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz, 32 GB RAM, 1 NVIDIA 1080Ti GPU of 12 GB memory.

C. Experimental Results

1) Result for Answering RQ1: To identify whether our framework is able to detect multiple types of smart-contract vulnerabilities, we evaluate DL models that capsuled in framework on reentrancy and time dependence vulnerabilities datasets, respectively. And the effectiveness is reported in TABLE I.

In this group of experiments, we first convert all the statements in a function into vectors directly, which is our baseline method. It can be observed in TABLE I that all the eight DL models can detect both vulnerabilities to a certain extent, but the overall performance is poor with baseline method. Specially, the Bi-LSTM model performs better in reentrancy vulnerability dataset than other models in terms of all the four metrics, and it achieve F1 of 70.23%. Except for Bi-LSTM, the performances of the other seven models are relatively close, whose F1 are all around 50%. Additionally, as for *time dependence* vulnerability, all the models yield a F1 around 70% with baseline method, and the TextCNN model achieve the highest R of 84.54% while the Bi-LSTM model achieve the highest P of 83.04%.

In summary, the DL model can automatically learn semantic knowledge from the vector representation of Solidity source code to detect different types of vulnerabilities, especially *time dependence* vulnerability. One possible reason is that time

Authorized licensed use limited to: Japan Advanced Institute of Science and Tech. Downloaded on July 27,2024 at 08:37:48 UTC from IEEE Xplore. Restrictions apply.

dependence vulnerability samples have syntactic or semantic features that are easier to distinguish. This result proves the potential of DL application in smart contract vulnerability detection, and shows that the DL model has indeed learned some vulnerability characteristics.

2) Result for Answering RQ2: In order to answer whether our framework can be improved by incorporating VCS. We compare the DL model trained on VCS that are automatically extracted with the baseline model, to further illustrate the effectiveness of VCS.

As shown in TABLE I, it can be concluded obviously that our enhanced models perform superior to the baseline methods on detecting both types of vulnerabilities. The improvement in each of the metric is substantial: In the detection of the reentrancy vulnerability, the F1 of all models increased by 16.76% on average, and the F1 of the GRU model improved by 25.76%. What's more, Bi-LSTM model achieves the highest F1 of 86.87%. In the detection of the *time dependence* vulnerability, the F1 of all models increased by 5.43% on average, and the F1 of the LSTM model improved by 14.06%. Among these models, LSTM-Attention model achieves the highest F1 of 79.93%.

One can see that both the convolutional neural network and recurrent neural network models improve significantly after training on the VCS. This result proves that VCS can improve the effectiveness of our framework, especially the overall effectiveness in F1. The intuition behind results shows that data flow and control flow in source code are able to help models concentrate on key point of vulnerability and learn vulnerable patterns better. Additionally, we find that VCS improves the performance of detecting reentrancy vulnerability much better than time dependence vulnerability. We think the reason is that reentrancy vulnerability samples have richer data flow and control flow relationships, while time dependence vulnerability samples usually have relatively simpler data flow and control flow relationships. Therefore, the improvement brought by VCS is smaller.

3) Result for Answering RQ3: To evaluate the worthwhileness of our framework, we selected state-of-the-art smart-contract vulnerability detection methods from two categories for comparison. First, we select learning-based SOTA approach [15], which applies graph neural networks to detect reentrancy and time dependence vulnerabilities. Next, we select another six methods, including Smartcheck, Securify, Mythril, Oyente, Slither and Osiris, while these detection approaches do not use DL models.

TABLE II shows the quantitative results, and we have the following observations. Among the six traditional methods without DL phase, Securify obtains a 55.10% F1 on reentrancy vulnerability detection and Mythril obtains a 42.48% F1 on timestamp dependence vulnerability detection, which is quite low. This stem from the fact that these methods detect these two kinds of vulnerabilities by crudely checking whether the statements contains *call.value*/*block.timestamp* or not. The method using graph neural networks has better performance in terms of all the four metrics, which gains a 78.84% F1 on reentrancy vulnerability detection and 79.08% F1 on *timestamp dependence* vulnerability detection.

In comparison, our framework outperforms above state-of-the-art methods by a large margin. The Bi-LSTM model in our framework achieves a F1 of 86.87% at detecting reentrancy vulnerability, gaining a 8.76% F1 improvement over graph-neural-network based method and a 31.77% F1 improvement over the Securify, which performs the best among traditional methods. Moreover, the LSTM-Attention model obtains a F1 of 79.93% at detecting *timestamp dependence* vulnerability, which is higher than the above-mentioned seven methods. The evidences reveal the great potential of applying our framework to smart-contract vulnerability detection.

VI. Conclusion

In this paper, we propose a modularized DL-based framework to fully automatically detect reentrancy and time dependence vulnerabilities in smart contracts, called DeeSCVHunter. Additionally, we innovatively propose a notion of Vulnerability Candidate Slice (VCS). VCS contains rich data and control dependencies between program elements, which aims at helping DL models focus on the key point of vulnerability and capture more semantic information. Extensive experiments prove the effectiveness of VCS. And results show that our framework significantly outperforms other seven state-of-the-art methods. It should be noted that we have made DeeSCVHunter publicly available, and researchers can conveniently establish a smart contract vulnerability detection system for evaluation or applying in actual situation.

Acknowledgment

This research was supported by the National Nature Science Foundation of China under Grant No. 61941116, National Nature Science Foundation of China under Grant No. U1936119 and National Key R&D Program of China under Grant No.2019QY(Y)0602.

References

[1] Tsankov, Petar, et al. "Securify: Practical security analysis of smart contracts." Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018.

[2] Sankar, Lakshmi Siva, M. Sindhu, and M. Sethumadhavan. "Survey of consensus protocols on blockchain applications." 2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS). IEEE, 2017.

[3] Clack, Christopher D., Vikram A. Bakshi, and Lee Braine. "Smart contract templates: foundations, design landscape and research directions." arXiv preprint arXiv:1608.00771 (2016).

[4] Buterin, Vitalik. "Ethereum: A next-generation smart contract and decentralized application platform." URL https://ethereum.org/en/whitepaper/ (2014).

[5] Hinkes, Drew. "A legal analysis of the DAO exploit and possible investor rights." Bitcoin Magazine (2016).

[6] Slowmist Hacked. Available: https://hacked.slowmist.io/en/ (2018)

[7] Feist, Josselin, Gustavo Grieco, and Alex Groce. "Slither: a static analysis framework for smart contracts." 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). IEEE, 2019.

[8] Luu, Loi, et al. "Making smart contracts smarter." Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. 2016.

[9] Mueller, Bernhard. "Smashing ethereum smart contracts for fun and real profit." HITB SECCONF Amsterdam (2018).

Authorized licensed use limited to: Japan Advanced Institute of Science and Tech. Downloaded on July 27,2024 at 08:37:48 UTC from IEEE Xplore. Restrictions apply.

[10] Tikhomirov, Sergei, et al. "Smartcheck: Static analysis of ethereum smart contracts." Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain. 2018.

[11] Tsankov, Petar, et al. "Securify: Practical security analysis of smart contracts." Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018.

[12] Jiang Bo, Ye Liu, and W. K. Chan. "Contractfuzzer: Fuzzing smart contracts for vulnerability detection." 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2018.

[13] Liu, Chao, et al. "Reguard: finding reentrancy bugs in smart contracts." 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion). IEEE, 2018.

[14] Perez, Daniel, and Benjamin Livshits. "Smart contract vulnerabilities: Does anyone care?." arXiv preprint arXiv:1902.06710 (2019).

[15] Zhuang Yuan, et al. "Smart Contract Vulnerability Detection Using Graph Neural Networks." IJCAI. 2020.

[16] Nikolić, Ivica, et al. "Finding the greedy, prodigal, and suicidal contracts at scale." Proceedings of the 34th Annual Computer Security Applications Conference. 2018.

[17] Torres, Christof Ferreira, Julian Schütte, and Radu State. "Osiris: Hunting for integer bugs in ethereum smart contracts." Proceedings of the 34th Annual Computer Security Applications Conference. 2018.

[18] Samreen, Noama Fatima, and Manar H. Alalfi. "Reentrancy Vulnerability Identification in Ethereum Smart Contracts." 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE). IEEE, 2020.

[19] Huang, TonTon Hsien-De. "Hunting the ethereum smart contract: Color-inspired inspection of potential attacks." arXiv preprint arXiv:1807.01868 (2018).

[20] Tann, Wesley Joon-Wie, et al. "Towards safer smart contracts: A sequence learning approach to detecting security threats." arXiv preprint arXiv:1811.06632 (2018).

[21] Mikolov, Tomas, et al. "Efficient estimation of word representations in vector space." arXiv preprint arXiv:1301.3781 (2013).

[22] Bojanowski, Piotr, et al. "Enriching word vectors with subword information." Transactions of the Association for Computational Linguistics 5 (2017): 135-146.

[23] Pennington, Jeffrey, Richard Socher, and Christopher D. Manning. "Glove: Global vectors for word representation." Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). 2014.

[24] Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." arXiv preprint arXiv:1409.0473 (2014).

[25] Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.

[26] Kim, Yoon. "Convolutional neural networks for sentence classification." arXiv preprint arXiv:1408.5882 (2014).

[27] Vinyals, Oriol, et al. "Show and tell: A neural image caption generator." Proceedings of the IEEE conference on computer vision and pattern recognition. 2015.

[28] Zhou, Peng, et al. "Attention-based bidirectional long short-term memory networks for relation classification." Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers). 2016.

[29] Chollet, F., et al.: Keras. https://github.com/fchollet/keras.

[30] Abadi, Martín, et al. "Tensorflow: A system for large-scale machine learning." 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16). 2016.

[31] Mané, D. "TensorBoard: TensorFlow's visualization toolkit, 2015."

[32] Horwitz, Susan, Thomas Reps, and David Binkley. "Interprocedural slicing using dependence graphs." ACM Transactions on Programming Languages and Systems (TOPLAS) 12.1 (1990): 26-60.

[33] Sinha, Saurabh, Mary Jean Harrold, and Gregg Rothermel. "System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow." Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002). IEEE, 1999.

[34] ANTLR, 2020. http://www.antlr.org/.

[35] B. Mueller. [n. d.]. Mythril. https://github.com/ConsenSys/mythril.

Authorized licensed use limited to: Japan Advanced Institute of Science and Tech. Downloaded on July 27,2024 at 08:37:48 UTC from IEEE Xplore. Restrictions apply.