Smart Contract Vulnerability Detection Using Code Representation Fusion

Ben Wang[1], Hanting Chu[1], Pengcheng Zhang[1], Hai Dong[2]
[1]College of Computer and Information, Hohai University, Nanjing, China
[2]School of Computing Technologies, RMIT University, Melbourne, Australia
shawnwb@163.com; 1354868944@qq.com; pchzhang@hhu.edu.cn; hai.dong@rmit.edu.au

***Abstract*—At present, most smart contract vulnerability detection use manually-defined patterns, which is time-consuming and far from satisfactory. To address this issue, researchers attempt to deploy deep learning techniques for automatic vulnerability detection in smart contracts. Nevertheless, current work mostly relies on a single code representation such as AST (Abstract Syntax Tree) or code tokens to learn vulnerability characteristics, which might lead to incompleteness of learned semantics information. In addition, the number of available vulnerability datasets is also insufficient. To address these limitations, first, we construct a dataset covering most typical types of smart contract vulnerabilities, which can accurately indicate the specific row number where a vulnerability may exist. Second, for each single code representation, we propose a novel way called AFS (AST Fuse program Slicing) to fuse code characteristic information. AFS can fuse the structured information of AST with program slicing information and detect vulnerabilities by learning new vulnerability characteristic information.**

***Index Terms*—vulnerability detection, deep learning, code representation fusion, AST, program slicing**

I. Introduction and Motivation

Smart contracts are autonomous programs that run on the blockchain, which are susceptible to coding threats due to the design issues of the blockchain, Ethereum and Solidity. Many scholars have developed tools based on traditional vulnerability detection techniques for smart contract vulnerability detection [1]. A common limitation **(problem 1)** with this type of approaches is that they use *manually-defined* patterns to detect vulnerabilities. Although these approaches achieve certain progress in detecting vulnerabilities, their accuracy rates are relatively low [2]. Vulnerability detection based on deep learning usually achieves high accuracy and completeness [5]. A common limitation (**problem 2**) for this category of detection approaches is that they mostly rely on a single form of code representation, such as code tokens, AST, control flow graphs, etc, which may not contain rich semantic and syntactic code features. Inspired by [2], we aim to explore new applications based on fused features for smart contract vulnerability detection. The 3rd pending issue in this area (**problem 3**) is the shortfall of accurately labelled and multi-vulnerability based public smart contract datasets for vulnerability detection.

To address the above problems, we propose an approach called **AFS** to automatically detect vulnerabilities in Solidity based smart contracts, instead of refined rules for matching vulnerability templates. The contributions of this paper are: (a) we combine program slicing information and structured information of AST to capture more semantic information of the vulnerabilities; (b) we aim to create an expandable and up-to-date real dataset with typical types of manually labeled vulnerabilities, including integer overflow and underflow, reentrancy, time-stamp dependency, suicide contracts, etc. This dataset is expected to be used for researchers to accurately and completely evaluate the performance of existing smart contract vulnerability detection approaches.

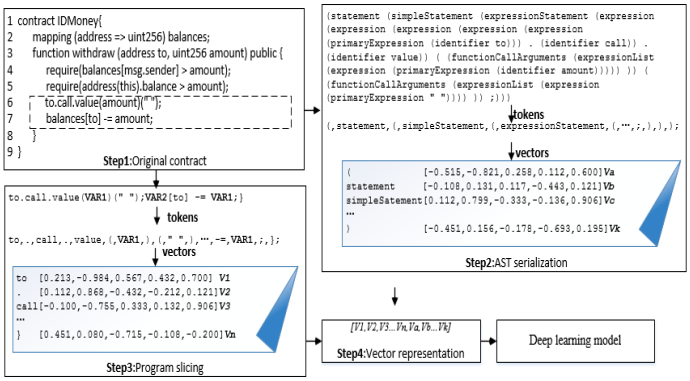

Fig. 1. Flowchart of the AFS data processing

II. Proposed Approach

We briefly describe the process of data collection before formally introducing the proposed vulnerability detection approach. The source of the dataset is twofold: First, we used the keywords of *smart contract vulnerability, vulnerable smart contracts*, and *smart contracts defects* to search smart contracts on GitHub and Gitter chat room (https://gitter.im/orgs/ethereum/rooms/); Second, we used Karl (https://github.com/cleanunicorn/karl) to collect new smart contracts from the Ethereum blockchain website (https://etherscan.io), where Karl is a tool used together with Mythril [1] for real-time blockchain monitoring. They can provide addresses of smart contracts that may cause vulnerabilities. For data labelling, we adopted Mythril, Slither, Oyente and SmartCheck [1] to detect the collected contracts. When 3/4 of the tools report that there is a problem with a certain line of a contract, we conduct a manual verification and label the vulnerability once it is verified. We then add the labelled contract into the vulnerability dataset.

B. Wang, H. Chu, P. Zhang and H. Dong, "Smart Contract Vulnerability Detection Using Code Representation Fusion," 2021 28th Asia-Pacific Software Engineering Conference (APSEC), Taipei, Taiwan, 2021, pp. 564-565, doi: 10.1109/APSEC53868.2021.00069.

AFS adopts a supervised learning paradigm, the process of which includes data processing, model training and model prediction. Among them, data processing is the most important stage. Fig 1 illustrates the main data processing steps for handling reentrancy vulnerability, where each box shows the result of the processing. The characteristic of a reentrancy vulnerability is that a contract uses *call-statements* to deposit ethers into a contract and then repeatedly controls the call-statements to deduct tokens held by the payee address [3]. This would trigger multiple invocations of the *call-statements* and even drain the deposit contract. Data processing is divided into 4 steps. **First**, we analyze the global function-call graph generated by Slither to extract a complete function-call path from a contract. For vulnerable contracts, we extract paths containing *call-statements* in the collected dataset. For non-vulnerable contracts, the deposit paths in which a function is declared as *payable* will be extracted. We then extract the corresponding complete functional fragment according to the function paths. **Second**, to obtain the structured information of the functional fragments, we adopt ANTLR (https://github.com/solidity-parser/antlr) (ANother Tool for Language Recognition) to parse those function fragments into AST (Abstract Syntax Tree). ANTLR is a powerful parser generator for translating structured text and analysing language. AST is a tree-like representation of abstract syntax structure of source code. We then deploy depth-first search to serialize the AST of the function fragments. In Figure 1, the sample result of the AST serialization on line 6-7 of a functional fragment **withdraw** is illustrated. The AST serialization can preserve the structured information of vulnerabilities. **Third**, program slicing techniques are applied to extract the semantic information of the code. Specifically, for reentrancy vulnerability, we first extract the *address* variable in front of the *call-statements* based on the complete function call path in step 1, and all the statements that operate on the *address* variable to form program slices. In Figure 1, for *address* variable **to**, line 3,6,7 will be added to the program slices. Then, we clean up the slices to remove some semantic-irrelevant information. For instance, we standardize variable names as 'VAR' for user-defined variable names. We then tokenize the AST information and the sliced code information and enter them straightly into the word2vec model for training two respective word vectors. Specifically, we set the length of the function fragment to be 100 through clustering, and the dimension of each token is searched in [100, 150]. **Fourth**, we concatenate these two word vectors as a new fusion vector. The new vector is a code representation combining these two sets of feature information. Finally, the concatenated vector is utilized as the input to train a deep learning based vulnerability detection model.

We plan to conduct the evaluation by comparing the proposed approach with the traditional vulnerability detection approaches and deep learning based approaches based on single code representations [4]. The comparison will be performed in 1) commonly used detection metrics, e.g. Accuracy, Recall, F1, etc, 2) detection speed and 3) capability of the approaches on learning undefined vulnerability types. The scalability of these approaches will also be tested in terms of these metrics.

III. Evaluation

We evaluate the performance of the approach on reentrancy vulnerability detection. Based on the proposed data collection approach, we analyzed 20,000 smart contracts, of which 1,832 (531,049 lines) meet the above requirements in step 1. We then used 1,832 smart contracts to conduct experiments, of which 207 (101,497 lines) contain reentrancy vulnerability. We divided them into a training set (80%) and a testing set (20%). We adopted LSTM, BLSTM-ATT [4] to evaluate the testing set. We also compared AFS with the method proposed in [4]. For each model, we repeat the experiments 10 times to calculate the average values. The experimental results are shown in TABLE 1. The aforementioned models are performed with learning rate = 0.002 (decaying every 10 epochs with a factor of 0.2), dropout rate = 0.5, batch size = 32, and the dimension of each token vector = 100.

TABLE I
Performance evaluation for LSTM, BLSTM-ATT

Model \ Metrics	ACC(%)	FP(%)	FN(%)	Recall(%)	F1(%)
LSTM(AST)	85.42	10.91	18.22	81.78	84.80
LSTM(AFS)	89.76	10.99	9.50	90.50	89.81
BLSTM-ATT([4])	90.48	9.85	9.13	90.87	90.56
BLSTM-ATT(AFS)	93.07	8.48	5.40	94.60	93.21

As shown in Table 1, in the LSTM model, AFS outperforms AST based on a single representation. AFS also performs well in BLSTM-ATT model, with 93.07% accuracy on the testing datasets, which can prove the effectiveness of AFS. It is worth noting that we currently only conduct experiments on a small number of reentrancy datasets. We will increase the amount of experimental data and apply AFS to other vulnerabilities in the future. Meanwhile, we will carry out the scheduled evaluation so as to prove the effectiveness of the proposed method.

Acknowledgements

The work is supported by the Natural Science Foundation of Jiangsu Province (No.BK20191297).

References

[1] Durieux T, Ferreira J F, Abreu R, et al. Empirical review of automated analysis tools on 47,587 ethereum smart contracts[C]//Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 2020: 530-541.

[2] Zou D, Wang S, Xu S, et al. μVulDeePecker: A deep learning-based system for multiclass vulnerability detection[J]. IEEE Transactions on Dependable and Secure Computing, 2019.

[3] Zhang P, Xiao F, Luo X. A framework and dataset for bugs in ethereum smart contracts[C]//2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2020: 139-150.

[4] Qian P, Liu Z, He Q, et al. Towards automated reentrancy detection for smart contracts based on sequential models[J]. IEEE Access, 2020, 8: 19685-19695.

[5] Li Y, Huang CL, Wang ZF, Yuan L, Wang XC. Survey of software vulnerability mining methods based on machine learning. Ruan Jian Xue Bao/Journal of Software, 2020,31(7):20402061 (in Chinese). http://www.jos.org.cn/1000-9825/6055.html