© 2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

A Generative Adversarial Networks-Based Integer Overflow Detection Model for Smart Contracts

Hao Li
School of Electrical and Data Engineering
University of Technology Sydney
Sydney, Australia
hao.li-7@student.uts.edu.au

Xu Wang
School of Electrical and Data Engineering
University of Technology Sydney
Sydney, Australia
xu.wang-1@uts.edu.au

Guangsheng Yu
Data61
CSIRO
Sydney, Australia
saber.yu@data61.csiro.au

Wei Ni
Data61
CSIRO
Sydney, Australia
wei.ni@data61.csiro.au

Ren Ping Liu
School of Electrical and Data Engineering
University of Technology Sydney
Sydney, Australia
renping.liu@uts.edu.au

***Abstract*—Due to the rapid development of blockchain technology in recent years, smart contracts have been widely applied in critical fields such as finance, insurance, healthcare, and the Internet of Things. However, smart contracts face increasingly serious security issues due to their unique operating environment and programming characteristics. We focus on Ethereum-based smart contracts and propose a high-precision and versatile detection method to address the integer overflow vulnerability, which significantly affects smart contract development and execution. Our method can also solve the problem of possible data shortage. Specifically, we utilize code embedding algorithms to convert Solidity-compiled smart contracts into spatial vectors, thereby retaining as much syntax and semantic information as possible. Based on this, we use Generative Adversarial Network (GAN) to train a small sample vector dataset to generate a substantial number of synthetic datasets. Our proposed model combines GAN discriminator feedback and vector similarity analysis to identify smart contracts that contain integer overflow vulnerabilities.**

***Index Terms*—blockchain, smart contract, integer overflow, code embedding, generative adversarial network, vulnerability detection**

I. Introduction

Smart contracts have found extensive applications in critical domains such as finance, insurance, healthcare, and the Internet of Things [1], owing to the rapid growth of blockchain technology [2] in recent years. In various research works related to smart contracts, most of the work is focused on the relatively more mature smart contracts of the Ethereum platform [3]. However, the entire process of smart contracts, from generation to execution, faces different security threats and vulnerabilities [4]. For example, in 2016, the Decentralized Autonomous Organization (DAO) was attacked [5]. The attacker exploited a recursive call vulnerability in the splitting function, which allowed multiple token withdrawals with a single contract call, and moved the target main chain currency to a DAO child chain. Ultimately, this attack resulted in the separation of more than 3 million ether from the asset pool. In addition, in 2017, a bug in the wallet and Sol's multi-signature contract resulted in the theft of 150,000 ether from the Parity wallet [4]. In 2018, the Beauty Chain project experienced a security vulnerability that caused it to collapse instantly, with a total value of about 10 billion US dollars [4].

Integer overflow, one of the common vulnerabilities in smart contracts, should not be underestimated [6]. It refers to the situation in which the value of an integer exceeds the maximum range of its type in a smart contract, causing the value to be truncated and returned to the minimum value, resulting in a numerical error. Attackers can exploit this vulnerability to perform improper operations, such as tampering with data or stealing assets. For example, suppose a smart contract is used to store a user's balance. When the balance increases to its maximum value, adding further balance results in integer overflow, resetting the balance value to the minimum, resulting in an incorrect balance. Attackers can exploit this vulnerability to gain additional assets or perform other harmful operations.

Currently, there are some limitations to existing vulnerability detection methods. Symbolic execution, represented by Oyente [7], can precisely explore the execution path of a contract and even detect vulnerabilities by analyzing the dependencies between transactions. However, its efficiency and accuracy in detecting vulnerabilities in multi-layered datasets are low. Fuzzing, represented by sFuzz [8], can detect path-reachable contract vulnerabilities by utilizing runtime information. However, it cannot automate the testing of contracts without source code or invocation interface information.

Moreover, there are some difficulties in obtaining a large number of smart contract source codes that meet the experimental requirements. Firstly, research has shown that only 1% of smart contracts are open-source [9]. Secondly, obtaining a large amount of source code that meets practical requirements may require a significant amount of time and resources due to the limitations of the Ethereum network and the number of nodes [10]. Additionally, manual data screening is necessary

to ensure data quality and security. In some cases, obtaining real vulnerability data may involve privacy protection and legal compliance issues. It should be noted that insufficient experimental data may reduce the detection model's accuracy and generalization ability.

This paper proposes a data processing model based on code embedding and Generative Adversarial Network (GAN) [11] to address these challenges. This model effectively reduces data requirements and enhances detection accuracy. Specifically, we first obtain the vector representation of contract code using code2vec [12] and train a GAN model using the vector representation of contracts with integer overflow vulnerabilities. After the training, the GAN discriminator is used to detect the target contract and obtain preliminary feedback on the existence of vulnerabilities. Then, vector similarity detection is performed between the target contract and the vulnerability contracts. By using a progressive detection approach, our model can further improve detection accuracy and reliably identify potential vulnerabilities. Our main contributions are as follows:

- We propose a smart contract vulnerability detection model that combines code embedding and GAN, which can effectively identify integer overflow vulnerabilities.
- We use the GAN model to generate a large amount of synthetic contract vector data, which enables us to achieve deep learning on small-sample data, alleviating the problem of difficulty in obtaining smart contract source code.
- We conduct experiments on 150 public Ethereum contracts, and the results show that our model achieves good scalability, high accuracy and efficiency.

The remaining sections of this paper are organized as follows. We conduct a survey of related work in Section II. Section III outlines the methodology of the detection model. In Section IV, we conduct experiments and analyze the experimental results. Finally, we conclude this paper in Section V.

II. Related Works

In the face of vulnerabilities that pose a serious threat to the execution and credibility of smart contracts, researching reasonable and effective detection solutions is of great importance. Automated vulnerability discovery is an important research field in software vulnerability discovery [13]. Smart contracts, as a new type of decentralized application, have many differences from traditional programs in terms of the runtime environment, lifecycle, and program characteristics. This situation also presents unique challenges for automated vulnerability discovery in smart contracts. A large amount of existing research focuses on how to apply existing techniques to achieve better-automated vulnerability detection on smart contracts. The mainstream detection methods can be classified into five types: Formal Verification [14], Symbolic Execution [7], Program Analysis [15], Taint Analysis [16], and Fuzzing Techniques [8]. This section will focus on analyzing Symbolic Execution and Fuzzing Techniques.

A. Symbolic Execution

Symbolic execution is a traditional technique for automated vulnerability detection, which is currently widely used in smart contract vulnerability detection. Its engine can provide a symbolic virtual execution environment for the target code, abstracting the external inputs required by the program as symbolic values with no fixed values, and exploring program branches by continuously solving path constraints. The main idea of this method is to convert uncertain inputs in the program execution process into symbolic values to promote program execution and analysis. Both smart contract programs and traditional programs can be abstracted as an execution tree. In the normal execution flow, each condition can obtain a definite answer because the program input is a fixed value, so only one branch is explored. However, in symbolic execution, the input value is an undefined symbolic variable. When encountering a conditional judgment, the symbolic execution engine will use a constraint solver to solve expressions that contain symbolic variables. Symbolic execution analyzes and records constraints in all solvable branches [7].

As a landmark symbolic execution tool, Oyente can automatically detect various security vulnerabilities in Ethereum smart contracts, including race conditions, timestamp dependencies, unchecked return values, reentrancy, integer overflow, and call stack overflow. Oyente constructs and supplements the control flow graph, uses the Z3 solver to solve the conditional jumps in the contract, determines the execution path based on the solving result, and collects contract information during the execution process to identify contract vulnerabilities. It can cover most of the contract code, discover various types of vulnerabilities, and improve the reliability and security of contracts [7]. However, it cannot handle some complex semantic features such as environmental variables, library functions, and dynamic calls; it can also not detect logical errors or functional defects such as permission control and algorithm correctness. Additionally, since symbolic execution is an NP-hard problem, it requires high computing resources and time.

B. Fuzzing Techniques

Fuzzing Techniques is a dynamic analysis technique that can detect vulnerabilities or anomalies in smart contracts by providing invalid, unexpected, or random data as input. Specifically, the execution process involves pre-processing the contract, including building a control flow graph and collecting function dependencies; then generating input data based on random mutation and coverage-based feedback strategies; executing the contract and collecting information on coverage, exceptions, state changes; and finally identifying contract vulnerabilities and further optimizing test cases [17].

sFuzz is one of the flagship tools of Fuzzing Techniques. It is a grey-box fuzzing tool based on AFL, which targets security vulnerabilities in Ethereum smart contracts. It generates transaction inputs according to the contract bytecode and executes them in the instrumented Ethereum Virtual Machine (EVM). By collecting feedback from the EVM, sFuzz can

detect vulnerabilities such as reentrancy, integer overflow, and call stack overflow. sFuzz combines the advantages of fuzz testing and symbolic execution, and can cover more code, discover vulnerabilities that are difficult to detect by black-box fuzzing, such as the vulnerabilities in the DAO contract and the Parity Wallet contract [8]. sFuzz also uses an adaptive seed selection strategy, dynamically constructs CFG, skips functions that do not change the state, and introduces mutation strategies for smart contracts. However, sFuzz may generate false positives and false negatives, and may not be able to handle some complex semantic features, and need to perform pre-processing or post-processing to improve testing efficiency and accuracy.

III. GAN-based Vulnerability Detection

To detect integer overflow vulnerabilities in smart contracts, we first collect code snippets that contain such vulnerabilities and use the code2vec tool to embed the code into vectors. Then, we construct a discriminator and a generator to judge and generate code snippets, respectively. These two models are trained with GAN, so that the discriminator can better distinguish between real and synthetic code snippets, and the generator can better generate synthetic code snippets with integer overflow vulnerability features. After GAN is trained, the generator produces a large amount of synthetic code to augment the vulnerable contract dataset that is used for subsequent similarity detection. Finally, we embed the target code snippets with code2vec and use the discriminator to predict them. If the discriminator outputs a positive class, it means that the code snippet may have an integer overflow vulnerability. Based on this, we further perform vector similarity detection [6] on the snippet and combine it with the discriminator result to determine whether there is a real vulnerability.

A. Data Generation

Fig. 1 illustrates the process of generating vulnerable contract data, which consists of three stages: code semantic analysis, code embedding, and synthetic code generation. As code2vec is designed to learn and vectorize Java source code by default, Solidity code must be pre-processed according to code2vec's training rules. First, we perform a lexical analysis on the Solidity code to break it into individual tokens. Next, we remove irrelevant information from the code, such as comments, whitespace, and blank lines, to clean the code and improve processing efficiency. Subsequently, all identifiers and literals are replaced with generic placeholders, such as "num," for ease of further processing. To capture the syntax of the code, we use the Solidity compiler to generate the Abstract Syntax Tree (AST) of the code, and assign a unique identifier to each node of the AST to differentiate between different node types. Finally, we generate all possible context paths using the node identifiers in the AST to represent the semantic information of the code [18].

After completing the semantic analysis, we train the code using code2vec to obtain corresponding vectors. To be specific,

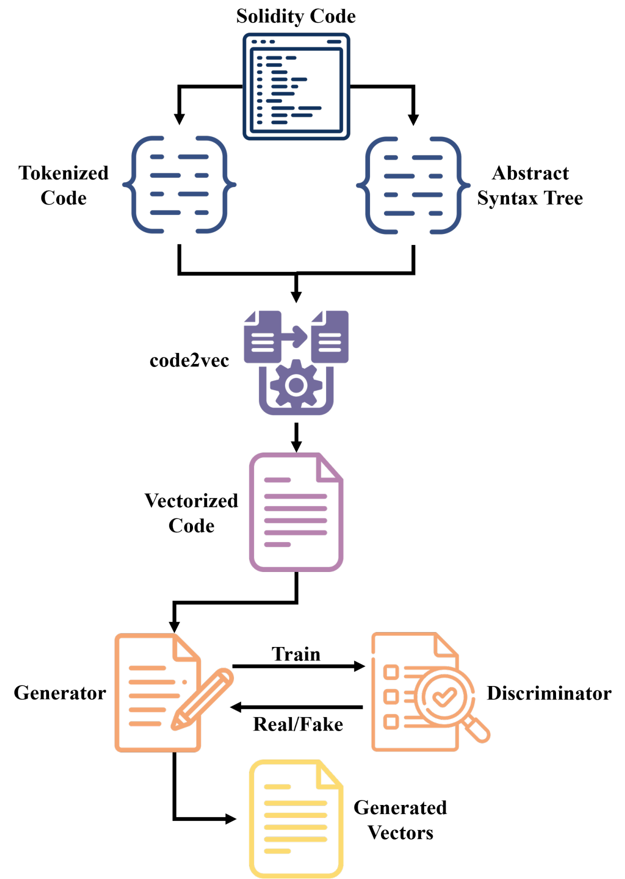

Fig. 1. Process of Data Generation.

code2vec is capable of extracting path and context features from the AST of code, which can be used to express relationships between code elements. A path consists of directed edges between two nodes, representing the syntactic structures passed through when accessing these nodes in the code, such as function calls and variable assignments. Context features provide additional information for describing code elements, such as their containing function and location. code2vec is able to convert paths and context features into corresponding vector representations, and concatenate them to express semantic information of code elements. Through this approach, code2vec generates a vector that can represent functions, variables, operators, and other code elements. Finally, code2vec connects all code element vectors to form a vector representation of the entire code segment. Here, the following formula is used to perform code embedding:

$$c = \sum_{i=1}^{n} a_i p_i; \tag{1}$$

$$p_i = H(s_i, t_i, r_i); \tag{2}$$

In this formula, c represents the vector representation of the code snippet, n is the number of AST paths, a_i represents the attention weight, p_i represents the vector representation of the

i-th AST path, s_i and t_i represent the vector representations of the start and end points of the path, r_i represents the vector representation of the node type sequence in the path, H is the neural network function.

Finally, we import the obtained vector dataset into a GAN for training to generate substantial synthetic data. The GAN model consists of a generator and a discriminator. In our model training process, the generator generates synthetic Solidity code vectors, while the discriminator differentiates between synthetic and real code vectors. Specifically, the generator takes random noise as input and generates synthetic Solidity code vectors by learning the distribution of actual Solidity code vectors. The discriminator takes two inputs, the synthetic and real Solidity code vectors, and outputs a value indicating whether the input code vector is synthetic or real. During the training process, the generator gradually learns how to generate synthetic vectors that are close to the real Solidity code vectors, while the discriminator gradually learns how to distinguish between synthetic and real vectors. The loss functions for the generator and the discriminator are as follows:

$$L_G = -E_{z \sim p(z)}[\log D(G(z))] \tag{3}$$

$$\begin{aligned} L_D = -E_{x \sim p_{data}(x)}[\log D(x)] + \\ E_{z \sim p(z)}[\log(1 - D(G(z)))] \end{aligned} \tag{4}$$

L_G is the loss function for the generator, L_D is the loss function for the discriminator, G is the generator, D is the discriminator, x is a real statement, z is random noise, $p_{data}(x)$ is the distribution of real statements, and $p(z)$ is the distribution of random noise. We can consider that GAN has finished training when the generator and the discriminator reach the Nash equilibrium [19]. In the completed training process, the generator will be able to generate synthetic Solidity code vectors that are similar to real Solidity code vectors. These vectors can be regarded as synthetic contract vectors. In this way, we can generate a large number of synthetic contract vectors to augment the vulnerable contract dataset. The expanded vulnerable contract dataset will be used for subsequent vector similarity detection.

B. Similarity Detection

Fig. 2 illustrates the process of vulnerability detection, which can be divided into two parts: GAN discriminator analysis and vector similarity detection. During the GAN training process, only contract vectors containing integer overflow vulnerabilities are used as inputs. Therefore, the discriminator trained by GAN can not only distinguish real and fake contracts but also detect whether the target contract contains integer overflow vulnerabilities. Specifically, the target contract is converted into a vector and is input into the discriminator. If the discriminator returns True, it means that the contract may contain integer overflow vulnerabilities.

If the GAN discriminator returns True, we will further examine the similarity between the target contract and the vulnerable contract set. To achieve automated detection, we consider using vector similarity as one of the detection criteria [20]. In the data generation phase, we use code2vec as a pre-processing method to obtain contract vectors that contain both the structural and semantic information of the source code. Based on these vectors, vector similarity can more accurately and quickly determine the similarity between Solidity codes. Specifically, cosine similarity [6] and correlation coefficient [21] are used to calculate the vector similarity between the target and vulnerable contracts. The specific formula is as follows:

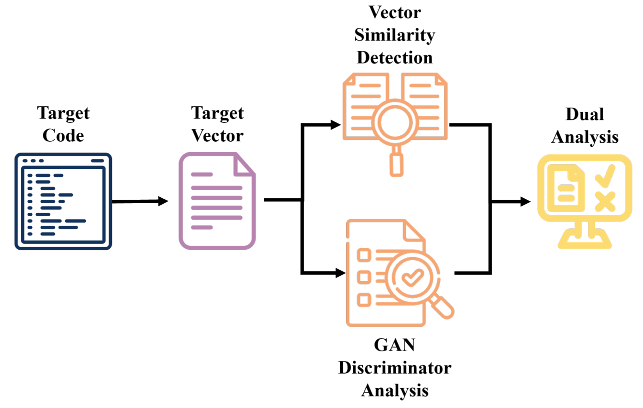

Fig. 2. Process of Similarity Detection.

$$\cos(x, y) = \frac{1}{n} \sum_{i=1}^{n} \frac{x \cdot y_i}{\|x\| \|y_i\|} \tag{5}$$

$$r = \frac{1}{n} \sum_{i=1}^{n} \frac{\sum_{j=1}^{m} (x_j - \bar{x})(y_{ij} - \bar{y}_i)}{\sqrt{\sum_{j=1}^{m} (x_j - \bar{x})^2 \sum_{j=1}^{m} (y_{ij} - \bar{y}_i)^2}} \tag{6}$$

where x represents the vector of the target contract, and y denotes the vector collection of the vulnerable contracts. Let y_i be the i-th vector in y, n be the cardinality of y, and m be the dimensionality of x. Furthermore, let $\bar{x}$ and $\bar{y}_i$ represent the mean values of the vectors, $\cos(x, y)$ denote the cosine similarity between two vectors, and r indicates the correlation coefficient.

When the cosine value and Pearson correlation coefficient approach 1, it indicates that the target contract is highly likely to have vulnerabilities. Here, we will calculate the weighted average of cosine similarity and Pearson correlation coefficient, and determine whether the target contract contains an integer overflow vulnerability based on a threshold. We will analyze and select appropriate weights and threshold in Section IV. By combining these two detection methods, we can significantly improve vulnerability detection accuracy.

IV. Experimental Result and Analysis

In this section, we will initially outline the experimental setup, followed by determining the essential parameters for

vector similarity through conducting experiments. Furthermore, we will compare and analyze our method with other existing schemes.

A. Experimental Setup

Our experiments are conducted on a Windows 10 (x64) computer equipped with an Intel Core CPU (2.30GHz×8), 16GB (3200MHz) memory, and Nvidia GeForce RTX 2060.

We collect 200 Solidity-compiled smart contracts from the Etherscan platform [22] and divide them into two datasets. The Training Set comprises 50 smart contracts, all containing integer overflow vulnerabilities and are used for training the GAN model. The remaining 150 smart contracts are classified as the Test Set, which is used for experimental analysis. Notably, the Test Set contains 70 contracts with integer overflow vulnerabilities, while the remaining 80 contracts are normal contracts. We generate 1950 synthetic contracts using the trained GAN model. These synthetic contracts are combined with 50 real contracts from the Test Set to form the vulnerable contract dataset, which is used for vector similarity detection.

B. Experimental Analysis

We use Accuracy in the confusion matrix [23] as the evaluation metric for the detection model:

$$ACC = (TP + TN)/(TP + TN + FP + FN) \quad (7)$$

TP represents the number of true positives, TN represents the number of true negatives, FP represents the number of false positives, and FN represents the number of false negatives. The accuracy of the detection model can be evaluated using the confusion matrix. The higher the accuracy (ACC), the higher the precision of the model.

Before the experiments, we define the formula for the vector similarity results as follows:

$$S = 0.5[\cos(x, y) \cdot W + r \cdot (1 - W)] \quad (8)$$

S is the result of vector similarity, W is the weight of cosine similarity, and $0 \leq W \leq 1$. When S is greater than or equal to the threshold value T ($0 \leq T \leq 1$), the target contract is judged to contain integer overflow vulnerabilities. When S is less than T, the target contract is judged not to contain an integer overflow vulnerability.

We first use experiments to determine the weight W of cosine similarity. We set the threshold to 0.85 and adjust the value of W to observe the change of model accuracy, as shown in Fig. 3. Through experiments, we set W to 0.74, at which the model achieves the highest accuracy.

After determining the value of W, we also need to determine the value of the threshold T through experiments. The experimental results are shown in Fig. 4. To balance the FP and FN rates, we set the threshold of our model to 0.9.

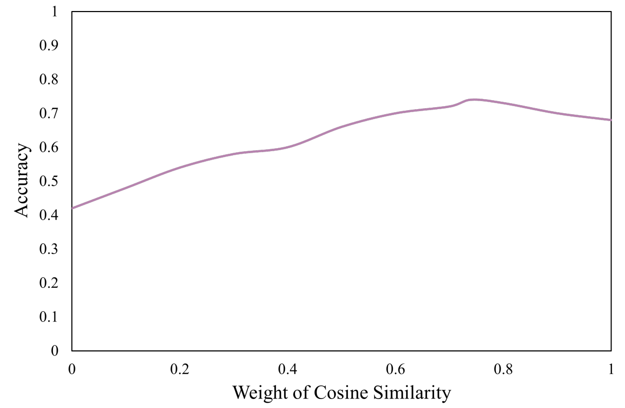

Fig. 3. Experiments on the Weight of Cosine Similarity.

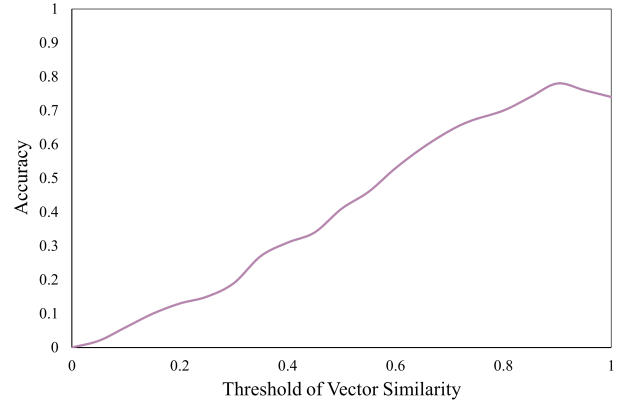

Fig. 4. Experiments on the Threshold of Vector Similarity.

C. Comparative Analysis

The Test Set is used for comparative experiments, which contains 70 contracts with integer overflow vulnerabilities and 80 contracts without vulnerabilities. To verify the detection ability of our model, we set up four control groups: one only using GAN discriminator analysis (Model α) and the other only using vector similarity detection (Model β), as well as the previously mentioned Oyente and sFuzz. The detection results are shown in Figure 5. Model α did not consider the similarity of contract vectors, and it missed some important structural and semantic information in the contract code, resulting in many FPs. Model β only detected based on the vector similarity between the target contract and the vulnerability contract set, without combining the GAN discriminator to pre-filter the contract code, so the detection accuracy was only 73%, and failed to reach the ideal detection level. Our model achieved a detection accuracy of 78%, second only to Oyente's 81%, and outperformed the other three control groups in terms of detection performance. As shown in Figure 6, we conduct experiments to measure the efficiency of methods by calculating the average detection time for each contract. It is obvious that our model has much higher detection efficiency than Oyente and sFuzz.

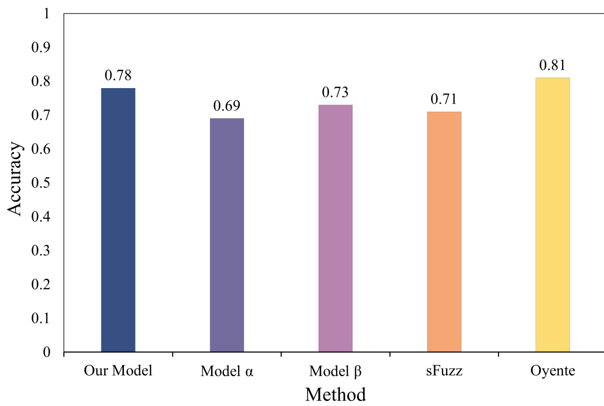

Fig. 5. Methods Accuracy Experiment.

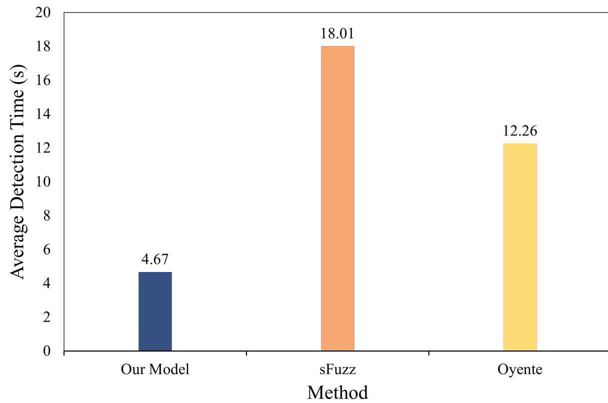

Fig. 6. Methods Efficiency Experiment.

V. Conclusion

In this paper, we proposed an intelligent contract vulnerability detection model based on code embedding and GAN. Our model combines the analysis of GAN discriminator feedback and code vector similarity to detect integer overflow vulnerabilities in smart contracts. By using GAN, we can generate a large amount of synthetic contract vector data, which can preserve the structural and semantic features similar to real contracts. GAN enables our model to achieve efficient vulnerability detection performance on small-sample data, effectively alleviating the data scarcity problem. Experimental results demonstrate the feasibility and effectiveness of the proposed model for detecting integer overflow in smart contracts. In future work, we can explore expanding the model to detect more types of vulnerabilities.

References

[1] X. Wang, X. Zha, W. Ni, R. P. Liu, Y. J. Guo, X. Niu, and K. Zheng, "Survey on blockchain for internet of things," *Computer Communications*, vol. 136, pp. 10–29, 2019.

[2] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, pp. 2–1, 2014.

[3] X. Wang, X. Zha, G. Yu, W. Ni, R. P. Liu, Y. J. Guo, X. Niu, and K. Zheng, "Attack and defence of ethereum remote apis," in *2018 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2018, pp. 1–6.

[4] D. He, Z. Deng, Y. Zhang, S. Chan, Y. Cheng, and N. Guizani, "Smart contract vulnerability analysis and security audit," *IEEE Network*, vol. 34, no. 5, pp. 276–282, 2020.

[5] C. Metz, "The biggest crowdfunding project ever–the dao–is kind of a mess," 2016.

[6] J. Huang, S. Han, W. You, W. Shi, B. Liang, J. Wu, and Y. Wu, "Hunting vulnerable smart contracts via graph embedding based bytecode matching," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 2144–2156, 2021.

[7] 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*, 2016, pp. 254–269.

[8] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

[9] 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: ESORICS 2017 International Workshops, DPM 2017 and CBT 2017, Oslo, Norway, September 14-15, 2017, Proceedings*. Springer, 2017, pp. 357–372.

[10] Z. Zhang, X. Wang, G. Yu, W. Ni, R. P. Liu, N. Georgalas, and A. Reeves, "A community detection-based blockchain sharding scheme," in *International Conference on Blockchain*. Springer, 2022, pp. 78–91.

[11] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[12] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[13] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *2012 fourth international conference on multimedia information networking and security*. IEEE, 2012, pp. 152–156.

[14] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy *et al.*, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, 2016, pp. 91–96.

[15] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, 2018, pp. 9–16.

[16] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *arXiv preprint arXiv:1812.05934*, 2018.

[17] P. Oehlert, "Violating assumptions with fuzzing," *IEEE Security & Privacy*, vol. 3, no. 2, pp. 58–62, 2005.

[18] C. Peng, S. Akca, and A. Rajan, "Sif: A framework for solidity contract instrumentation and analysis," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2019, pp. 466–473.

[19] H. Wang, J. Wang, J. Wang, M. Zhao, W. Zhang, F. Zhang, X. Xie, and M. Guo, "Graphgan: Graph representation learning with generative adversarial nets," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[20] G. Ma, N. K. Ahmed, T. L. Willke, and P. S. Yu, "Deep graph similarity learning: A survey," *Data Mining and Knowledge Discovery*, vol. 35, pp. 688–725, 2021.

[21] X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen, "Automated vulnerability detection in source code using minimum intermediate representation learning," *Applied Sciences*, vol. 10, no. 5, p. 1692, 2020.

[22] Etherscan, "Etherscan—the ethereum blockchain explorer," *https://etherscan. io/*, 2021.

[23] J. Liang, "Confusion matrix: Machine learning," *POGIL Activity Clearinghouse*, vol. 3, no. 4, 2022.