UTS UNIVERSITY OF TECHNOLOGY SYDNEY

Smart Contract Vulnerability Detection Based on Generative Adversarial Networks and Graph Matching Networks

by Hao Li

Thesis submitted in fulfilment of the requirements for the degree of

Master's by Research

under the supervision of Prof. Dr. Ren Ping Liu, and Dr. Xu Wang

University of Technology Sydney
Faculty of Engineering and IT

March, 2024

Certificate of Authorship / Originality

I, Hao Li, declare that this thesis is submitted in fulfilment of the requirements for the award of Master's by Research, in the School of Electrical and Data Engineering at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis. This document has not been submitted for qualifications at any other academic institution.

This research is supported by the Australian Government Research Training Program.

Signature: Production Note: Signature removed prior to publication.

Date: March 12, 2024

Thesis by Compilation Declaration				
Paper Title	**List of Authors**	**Current Status**	**Student's Contribution**	**Related Chapter**
A Generative Adversarial Networks-Based Integer Overflow Detection Model for Smart Contracts.	Hao Li, Xu Wang, Guangsheng Yu, Wei Ni, Ren Ping Liu.	Published in Proceedings of 2023 22nd International Symposium on Communications and Information Technologies (ISCIT).	Student's contribution is about 90%, comprising analytical work, simulations, and paper drafting. The co-authors provided feedback, comments, and technical support, which helped improve the paper.	Chapter-I Introduction; Chapter-II Literature Review; Chapter-III; Chapter-V Conclusion.
Smart Contract Vulnerability Detection Based on Generative Adversarial Networks and Graph Matching Networks.	Hao Li, Xu Wang, Guangsheng Yu, Wei Ni, Ren Ping Liu, Nektarios Georgalas, Andrew Reeves.	Accepted by the 2nd International Conference on Network Simulation and Evaluation 2023 (NSE).	Student's contribution is about 90%, comprising analytical work, simulations, and paper drafting. The co-authors provided feedback, comments, and technical support, which helped improve the paper.	Chapter-I Introduction; Chapter-II Literature Review; Chapter-IV; Chapter-V Conclusion.

	Name	Signature	Date
Student	Hao Li	Production Note: Signature removed prior to publication.	13.11.2023
Co-authors	Xu Wang	Production Note: Signature removed prior to publication.	19.11.2023
	Guangsheng Yu	Production Note: Signature removed prior to publication.	17.11.2023
	Wei Ni	Production Note: Signature removed prior to publication.	16.11.2023
	Ren Ping Liu	Production Note: Signature removed prior to publication.	17.11.2023
	Nektarios Georgalas	Production Note: Signature removed prior to publication.	13.11.2023
	Andrew Reeves	Production Note: Signature removed prior to publication.	13.11.2023

Abstract

With blockchain technology's decentralization and tamper-proof characteristics, smart contracts have developed rapidly and have been applied widely in some critical areas, e.g., the Internet of Things, digital management, healthcare, and finance. However, the security vulnerabilities of smart contracts have led to significant economic losses. Once deployed on the blockchain, smart contracts cannot be modified. Therefore, it is crucial to conduct pre-deployment vulnerability detection.

We propose a smart contract vulnerability detection model that combines code embedding and Generative Adversarial Networks (GAN), which can effectively identify integer overflow vulnerabilities. The study advances beyond traditional textual or structural analysis by exploring the Abstract Syntax Tree of smart contract source code for effective vectorization, while maintaining essential contract features. We use the GAN model to generate synthetic contract vector data, which facilitates the construction of the detection model with small-sample data and reduces the challenges in source code acquisition. Our method integrates GAN discriminator feedback and employs both cosine similarity and correlation coefficients for vector similarity analysis. It enhances the accuracy of vulnerability detection and has proved effective in practical scenarios.

Another novel detection method is proposed to identify reentrancy and integer overflow vulnerabilities, utilizing GAN and Graph Matching Networks (GMN). Expanding on prior research, we improve code representation by incorporating control and data flow from code functions and statements. This approach converts source code into a semantically and structurally rich contract graph, which preserves key contract features and outperforms traditional methods. We explore few-shot learning and use a graph-based GAN to overcome data starvation in training detection models. The innovative use of the GMN, an extension of Graph Neural Networks (GNN), enhances the efficiency of vulnerability detection. The novel GMN model uses a cross-graph attention mechanism to calculate the feature similarity between the target and the vulnerable contracts.

Our research not only enhances the precision and efficiency of vulnerability detection models, but also introduces new concepts of vector and graph embedding for machine learning-oriented representation of smart contract source code. Furthermore, the GAN-based model construction approach proposed in this thesis is advantageous for building high-performance detection models with limited data samples. Lastly, our study demonstrates that GNNs, exemplified by the GMN, have technical superiority and potential for further development, especially in learning and analyzing feature graphs of smart contract source code.

Acknowledgements

My Master's journey has passed in the blink of an eye, and with the passage of time, the traces of the twists and turns in the process have quietly faded away. However, I hope to use this thesis as a key to memory, ready to unlock those experiences that are worth reviewing and reflecting on, and evoke every thought, insight, and gain I had in my Master's journey.

Over the past almost three years, my research career has faced the huge challenge of the COVID-19 pandemic, which made me lose two years of traditional campus life. In these unusual days, I went through confusion, self-doubt, escape, and even was about to give up. But after deep reflection, I chose to fight hard for myself and my loved ones. Now looking back, I feel extremely grateful for my persistence. Carrying the strength of this experience, I am ready to continue on my future research journey.

In my Master's career, I would like to express my deep gratitude to my two supervisors, Prof. Ren Ping Liu and Dr. Xu Wang, without whose support and assistance my research topic would not have been possible. Prof. Liu is not only my academic mentor but also my life guide. He taught me self-affirmation and optimism for the future. At a critical moment in my career, he reached out to help me, for which I am grateful. Dr. Wang, on the other hand, played an indispensable role in guiding me through my research journey. When I first entered the field of research, I was intimidated by writing papers, but his guidance gave me confidence. Facing my struggles and stagnation, he never gave up on me, but gave me crucial guidance and encouragement when I made some progress, and even a timely push, which enabled my paper to develop from an extended abstract to a full paper. It is no exaggeration to say that without his help, there would be no thesis. To these two supervisors, I once again express my sincerest thanks.

I would like to thank my colleagues and co-authors, who played an indispensable role in my research journey. Prof. Wei Ni and Dr. Guangsheng Yu not only contributed countless valuable and constructive comments to my research, but also provided key revision suggestions for my papers. Without their strong support, I could not have completed those two important papers. In addition, I would also like to express my deep gratitude to Nektarios Georgalas and Andrew Reeves, who gave me great affirmation and support for my papers. At the same time, I would also like to thank all the colleagues in our research group for their care and support. I sincerely hope that all of you will successfully complete the challenge of obtaining a doctoral degree.

Here, I would like to especially and affectionately thank my parents and my girlfriend, who are the most important supporters in my life. In the past two years of hard times, especially in the days when I was slow and struggling, they must have endured unspeakable anxiety and bitterness in their hearts. However, they always supported me unswervingly and never gave up. Their perseverance became the source of strength for me to continue, letting me understand that since they did not give up on me, I had no reason to give up on myself. Fortunately, I did

not give up.

I will cherish this experience and move forward bravely in reflection and growth. With a deep memory of past challenges and growth, I embark on a new journey with gratitude and hope, and every step lights up the path to my future.

Hao Li
March 12, 2024
Sydney, Australia

Contents

1 Introduction **1**
1.1 Research Background . . . 1
1.1.1 Smart Contract Concepts . . . 1
1.1.2 Smart Contract Security . . . 3
1.1.3 Research Gap . . . 4
1.2 Motivation . . . 5
1.3 Research Contributions and Overview . . . 7
1.4 Thesis Structure . . . 9

2 Literature Review **11**
2.1 Introduction . . . 11
2.2 Research Background . . . 13
2.2.1 Blockchain . . . 13
2.2.2 Ethereum . . . 14
2.2.3 Smart Contract . . . 16
2.3 Smart Contract Vulnerabilities . . . 18
2.3.1 Code Level . . . 19
2.3.2 Ethereum Virtual Machine Level . . . 21
2.3.3 Blockchain Level . . . 22
2.4 Related Research on Vulnerability Detection . . . 23
2.4.1 Fuzzing . . . 23
2.4.2 Symbolic Execution . . . 27
2.4.3 Formal Verification . . . 29
2.4.4 Deep Learning . . . 31
2.4.5 Discussion . . . 34
2.5 Preliminaries . . . 37
2.5.1 Representation of Smart Contract . . . 37
2.5.2 Related Machine Learning Technologies . . . 40
2.6 Conclusion . . . 44

3 Detection Method Based on Code Representation and Generative Adversarial Networks **45**
3.1 Introduction . . . 45
3.2 Methodology Framework . . . 46
3.3 GAN-based Data Augmentation . . . 47
3.3.1 Code Preprocessing . . . 47
3.3.2 Code Embedding . . . 51

3.3.3 Code Generation . . . 52
3.4 Dual Similarity Detection . . . 53
3.4.1 GAN Discriminator Analysis . . . 54
3.4.2 Vector Similarity Analysis . . . 54
3.5 Experimental Result and Analysis . . . 55
3.5.1 Experimental Setup . . . 55
3.5.2 Dataset and Evaluation Metric . . . 56
3.5.3 Vector Similarity Parameter Experiment . . . 57
3.5.4 Detection Accuracy Experiment . . . 58
3.5.5 Detection Efficiency Experiment . . . 59
3.6 Conclusion . . . 61

4 Vulnerability Detection Method Based on Generative Adversarial Networks and Graph Matching Networks 62
4.1 Introduction . . . 62
4.2 Methodology Framework . . . 63
4.3 Graph Representation of Smart Contract . . . 64
4.3.1 Source Code Preprocessing . . . 64
4.3.2 Construction of Contract Graph . . . 66
4.4 GAN-based Graph Data Augmentation . . . 69
4.5 GMN-based Similarity Detection . . . 71
4.5.1 Graph Embedding . . . 71
4.5.2 Similarity Detection Model . . . 71
4.6 Experimental Result and Analysis . . . 75
4.6.1 Experimental Setup . . . 75
4.6.2 Dataset and Evaluation Metric . . . 76
4.6.3 GMN Parameters Experiment . . . 77
4.6.4 Detection Accuracy Experiment . . . 81
4.6.5 Detection Efficiency Experiment . . . 83
4.6.6 Method Scalability Experiment . . . 84
4.7 Conclusion . . . 88

5 Conclusions and Future Work 89
5.1 Summary of Outcomes . . . 90
5.2 Recommendations & Future Work . . . 90

6 Publication 93

List of Figures

1.1 Evolution Timeline of Smart Contracts. 2
1.2 Execution Mechanism of Smart Contracts. 3
1.3 Diagram of the Thesis Structure. 10

2.1 Overview of Path-Attention Network. 38
2.2 Example of One-Hot Encode. 39
2.3 Example of Skip-Gram. 40
2.4 Example of Continuous Bag-of-Word. 41
2.5 Structure of Generative Adversarial Networks. 42
2.6 Overview of Graph Matching Networks. 43

3.1 Generative Adversarial Networks-based Detection Method Framework. 47
3.2 Process of Data Generation. 48
3.3 Principle of Integer Overflow. 49
3.4 Abstract Syntax Tree of Smart Contract *Sol*. 50
3.5 Example of Abstract Syntax Tree Preprocessing. 51
3.6 Overview of Code Embedding. 52
3.7 Process of Synthetic Code Generation. 53
3.8 Overview of Similarity Detection. 53
3.9 Experiments on the Weight of Cosine Similarity. 58
3.10 Experiments on the Threshold of Vector Similarity. 59
3.11 Methods Accuracy Experiment. 60
3.12 Methods Efficiency Experiment. 60

4.1 Framework for Detection Method Based on Generative Adversarial Networks and Graph Matching Networks. 63
4.2 Process of Graph Representation. 65
4.3 Abstract Syntax Tree of Smart Contract RE. 67
4.4 Control Flow Graph of Smart Contract RE. 68
4.5 Data Flow Graph of Smart Contract RE. 68
4.6 Original Contract Graph of Smart Contract RE. 69
4.7 Contract Graph (CG) of Smart Contract RE. 69
4.8 Process of Graph Data Augmentation. 70
4.9 Overview of the Skip-Gram Model. 72
4.10 Construction and Workflow of Detection Model. 72
4.11 Example of Cross-graph Attention Mechanism. 74
4.12 Experiments on the Embedding Dimension. 78
4.13 Experiments on the Number of Hidden Layers. 79
4.14 Experiments on the Learning Rate. 79

4.15 Experiments on the Number of Iteration. 80
4.16 Experiments on the Similarity Threshold. 81
4.17 Methods Accuracy Experiment. 82
4.18 Methods Efficiency Experiment. 83
4.19 Method Computational Efficiency Experiment. 85
4.20 Method Data Requirement Experiment. 86
4.21 Method Testing Efficiency Experiment. 87

List of Tables

2.1 Classification of Smart Contract Vulnerabilities. 19
2.2 Classification of Smart Contract Vulnerability Detection Methods. 24

3.1 Example of Integer Overflow Features. 49
3.2 Summary of Original Dataset 56
3.3 Summary of Augmented Dataset 56
3.4 Confusion Matrices 57

4.1 Examples of Vulnerability Feature. 66
4.2 Control Structures of Solidity Code. 66
4.3 Summary of Original Datasets. 76
4.4 Summary of Augmented Datasets. 76
4.5 Summary of Datasets in Method Scalability Experiment. 85

Chapter 1

Introduction

This chapter initially offers a brief insight into the background of our research and delves deeply into the current state and security issues of smart contracts within the Ethereum environment. We discuss the current state of smart contract vulnerability detection research and analyze the limitations of existing research, thereby presenting and elaborating on the motivation for our research. In the final section of this chapter, we summarize the key contributions of our research and provide an overview of the structural organization of the entire thesis.

1.1 Research Background

1.1.1 Smart Contract Concepts

In 1994, computer scientist and cryptographer Szabo [1] initially coined the term "smart contract" to describe a digital form of promise. Essentially, a smart contract is an agreement or contract executed through programming code, designed to automatically fulfill its terms once the predetermined conditions agreed upon by all involved parties are satisfied. In practice, the content of the smart contract consists of program code. Developers just need to write the business logic code of the smart contract like programming, then store it in an appropriate blockchain system, and the contract function can take effect.

In 2008, cryptographer Nakamoto [2] created a virtual currency system called "Bitcoin" and officially proposed the concept of "blockchain". The related blockchain system was launched in 2009. He described the blockchain as a special data structure and an immutable and decentralized shared ledger. Subsequently, Buterin [3] integrated the concept of smart contracts into blockchain technology in the Ethereum white paper, significantly expanding the blockchain's application beyond merely currency-related fields and marking the advent of the Blockchain 2.0 era. Among the numerous blockchain systems that facilitate smart contracts, Ethereum stands out as the pioneering platform for their implementation and has since emerged as the largest blockchain platform globally. Thus, smart contracts offer a novel application method for blockchain that goes beyond financial currency, making it possible to manipulate underlying blockchain data programmatically. Ethereum, one of the major platforms applying blockchain technology, not only provides the function of digital currency, but also allows users to write smart contracts to manage underlying blockchain data, greatly expanding the application scope of blockchain and enabling various types of applications to operate on blockchain. Figure 1.1 provides a timeline of the evolution of smart contracts.

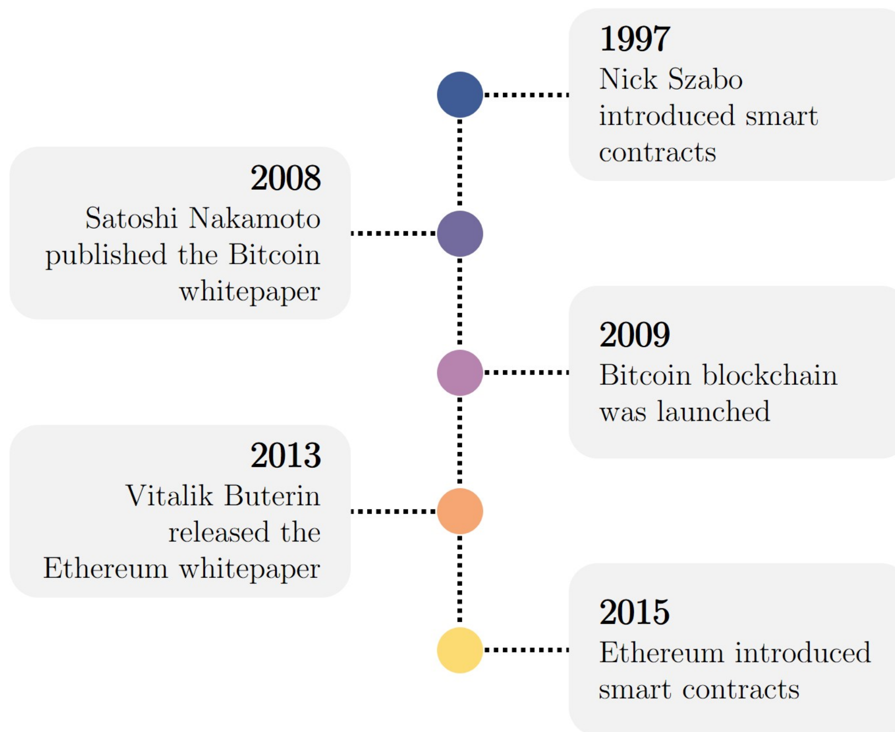


Figure 1.1: Evolution Timeline of Smart Contracts.

The execution mechanism of smart contracts, as illustrated in Figure 1.2, follows an automated process. Once all parties have agreed upon and signed the contract, it is encoded as programming code and subsequently created and stored on the blockchain. These contracts encapsulate a series of predefined states, transition rules, and conditions that trigger the execution of the contract. Upon fulfilling these conditions, the smart contract is activated within the blockchain network, where it undergoes verification by the nodes before being executed with the corresponding operations recorded on the blockchain. The execution status of smart contracts is continuously monitored by the blockchain, ensuring their accurate and precise execution when the triggering conditions are met [4]. Smart contracts have the following four characteristics:

1) Triggered by transactions, no human interaction is needed.
2) Once a smart contract is initiated, its execution cannot be halted.
3) In the blockchain network, every node is aware of the smart contract because its correctness must be verified by the majority of nodes.
4) It can be adjusted and optimized according to different scene requirements.

Over time, various industries have recognized the ease of use of smart contracts, and smart contracts have flourished in areas such as finance, management, healthcare, the Internet of Things, and supply chains. Smart contracts have a natural fit with the financial industry: their immutable, public, transparent, and decentralized characteristics avoid the introduction of a strong third party, the feature of peer-to-peer transactions simplifies transaction procedures, and the distributed ledger provided by blockchain offers convenience for financial regulatory authorities to trace and investigate all transaction behaviors, these features collectively ensure the safety of investors' funds. In addition, smart contracts are booming in the management field. The traceability of smart contracts makes them highly adaptable to activities such as election voting. It also has broad application prospects in digital asset copyright and business

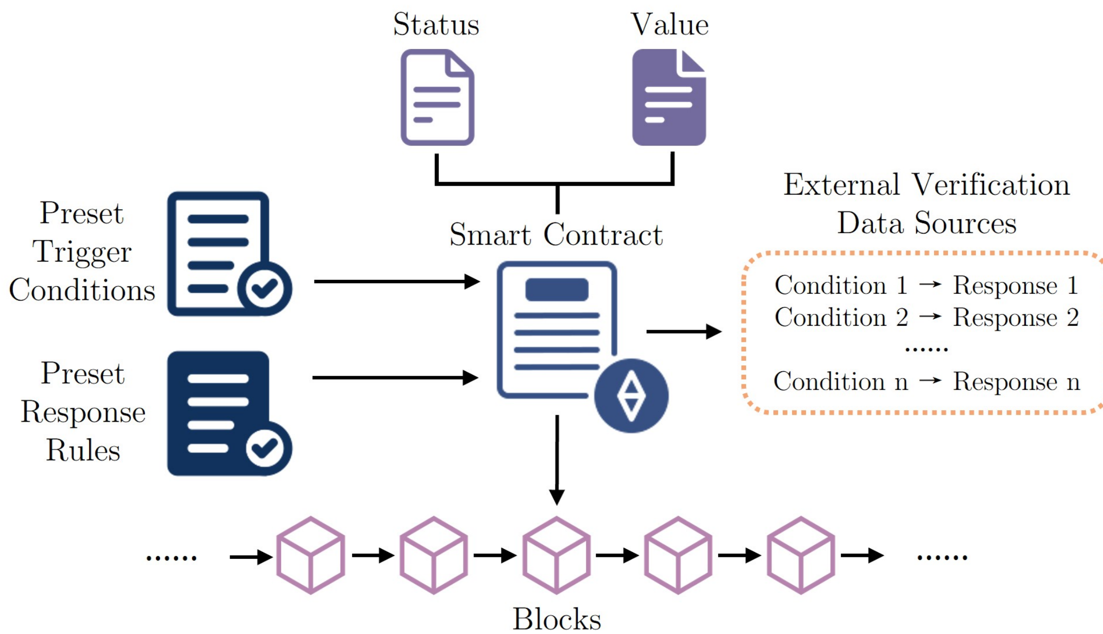


Figure 1.2: Execution Mechanism of Smart Contracts.

process management. One major application of smart contracts in the medical field is electronic medical records, which can set access permissions, allowing medical practitioners to have more granular access permissions to patient data, and preventing medical records from being leaked and tampered with. Furthermore, the convergence of IoT with decentralized smart contracts represents a significant trend in development. Smart contracts streamline intricate processes within IoT networks and offer solutions for enhancing resource sharing, thereby boosting industry efficiency, bolstering information security, and diminishing overall application costs.

1.1.2 Smart Contract Security

With the increasing application of Ethereum, its financial value has also grown, thereby highlighting the ever-growing security issues. To ensure stable system operation, Ethereum integrated a series of security measures in its initial design: cryptographic technology is used for data encryption and verification to ensure the safety of data in Ethereum; decentralized peer-to-peer networks and consensus algorithms are employed to prevent data loss or tampering from centralized hosts; and virtual machine technology is utilized to execute smart contracts in an isolated sandbox environment, thus limiting the impact of malicious smart contracts.

The original intention of smart contracts was to automate the execution of contract terms, reduce the dependence on trusted third parties, and improve the efficiency and security of transactions. However, as smart contracts are widely applied in various fields such as finance, supply chain, and the Internet of Things, their security issues have gradually become a research focus.

The security issues of smart contracts mainly stem from the complexity of their programming languages and the immutability of blockchain platforms. For example, smart contracts on the Ethereum platform, although providing a Turing-complete programming environment, also bring new security challenges. In 2016, the DAO project suffered a reentrancy attack [5], resulting in the theft of about 50 million US dollars, which not only caused huge economic

losses, but also exposed the vulnerability of smart contracts in the design and implementation process. Thereafter, the Parity wallet vulnerability incident further highlighted the harm of smart contract programming flaws. In 2017, the Parity multi-sig wallet contract was attacked due to design defects and negligence in the repair process, resulting in the theft or freezing of about 30 million US dollars worth of ether [6]. These events emphasized the necessity of conducting thorough security testing and continuous monitoring in smart contract development.

In 2018, Coincheck was hacked [7], resulting in the theft of 52.3 million NEM tokens, worth about 534 million US dollars. This attack exposed the security loophole of the exchange in the hot wallet management, and the hacker stole a large amount of funds through unauthorized transactions. Coincheck's incident stressed the importance of security measures in asset storage and protection, especially when interacting with smart contracts. The security of smart contracts directly affects the asset security of platforms such as exchanges, because the improper use or vulnerability of smart contracts may lead to the illegal transfer of funds.

The security issues of smart contracts are not limited to a single platform. With the globalization of blockchain technology, the security vulnerability problem of smart contracts has become a cross-border challenge. In August 2021, Poly Network, a cross-chain interoperability protocol, was hacked, resulting in the theft of more than 610 million US dollars worth of assets [8]. The attacker exploited a key vulnerability in the smart contract and successfully modified the Keeper role of EthCrossChainData.sol by constructing a specific cross-chain transaction, thereby controlling the asset transfer. This incident revealed the potential security risks of cross-chain protocols in the design and implementation process, especially in the complexity of cross-chain communication and contract interaction.

These major historical events constitute the background of smart contract security, which affected the development of blockchain technology and promoted the advancement of security testing technology. In this context, it is particularly important to conduct in-depth security testing and continuous monitoring of smart contracts, which is essential for maintaining the security and healthy development of the blockchain ecosystem.

1.1.3 Research Gap

It is significant to study reasonable and effective detection schemes in the face of vulnerabilities that pose serious threats to the execution and credibility of smart contracts. In recent years, a range of automated methods for detecting vulnerabilities have emerged, leveraging traditional techniques. These methods can be broadly categorized based on their underlying technical approaches: formal verification [9], symbolic execution [10], and fuzzing [11].

Fuzzing is a widely used dynamic vulnerability detection technique that works by generating a large number of random normal and abnormal inputs for the program during its normal execution, monitoring the program's state and behavior to detect vulnerabilities. Another common method for smart contract vulnerability detection is symbolic execution, which aims to assess whether the program works as expected. In simple terms, it simulates the execution process of the program using symbolic values. It finds the exact input values or value ranges that trigger vulnerabilities, thereby identifying potential vulnerabilities in the program. Formal verification can be divided into model checking and deductive verification. Model checking is a process of verifying whether a specific specification matches the model, using a finite state model to check whether the smart contract conforms to a specific state. If a contradiction arises with the specification, it could indicate a vulnerability. Deductive verification involves

establishing logical rules and using mathematical logic to demonstrate that a smart contract has specific properties.

Through analysis and experiments utilizing traditional methods, it has been observed that fuzzing faces limitations in automated testing and ideal path coverage for contracts lacking source code, and struggles with accurately pinpointing the location of vulnerabilities. Symbolic execution confronts challenges of path explosion as program complexity increases, as well as issues of timeouts or unsolvability in constraint resolution on complex paths, impacting its efficiency and accuracy. Formal verification necessitates the manual writing of specifications, which not only increases the workload and the probability of errors but also struggles to emulate the complete execution environment and random factors, affecting the accuracy and completeness of the verification. It is imperative to highlight that these traditional methods evaluate the security of smart contracts solely based on the execution process and outcomes. Such an approach not only diminishes the efficiency of vulnerability detection but also overlooks critical information inherent in the smart contract source code, leading to suboptimal accuracy in detection outcomes.

In response to the limitations of traditional methods in smart contract vulnerability detection, researchers have begun to delve deeper into the analysis of smart contract source code. This shift has led to the development of innovative methods focusing on textual analysis and extraction of critical information from the code. These methods, while pioneering, often struggle to encapsulate the multifaceted nature of smart contracts, leading to gaps in detection accuracy. One significant challenge lies in the singular approach to code representation, which may overlook complex interactions within the contracts. Additionally, the reliance on extensive training datasets for building these models increases the research burden and escalates development costs. The scarcity of comprehensive and diverse training samples further hinders the performance of these models. Another crucial aspect yet to be adequately addressed in current research is the development of advanced feature learning models, which are essential for improving the precision and effectiveness of vulnerability detection in smart contracts.

1.2 Motivation

Blockchain technology has always been a focus of attention for central banks and financial institutions both domestically and internationally. As a crucial part of blockchain applications, the use of smart contracts in various fields has made the automation of processes and payments more convenient, thereby enhancing the efficiency of global trading markets. However, the issue of smart contract security has always been a global concern. Durieux [12] used nine of the most advanced automated analysis tools to conduct a large-scale analysis of smart contracts. When evaluating vulnerabilities of the same category of smart contracts, they found significant differences in the results of different detection tools, and each vulnerability detection tool had a fairly high rate of missed detection, with only 42% of vulnerable smart contracts being identified by all detection tools. At the same time, when analyzing real-world contract datasets, they found that 97% of smart contracts were marked as vulnerable by these detection tools, indicating a large number of false positives in the results. Thus, it is clear that the current smart contract detection technology still falls short in terms of efficiency, scale, and scalability to meet the rapidly increasing audit needs of smart contracts.

The conventional manual vulnerability detection method can seem like a laborious and unrewarding task, requiring experienced security experts to inspect every contract for vulnerabilities,

making it inefficient meticulously. However, with the rapid development of artificial intelligence and machine learning, machine learning has demonstrated robust capabilities in classification and recognition problems [13]. In computer software security, numerous researchers have begun applying machine learning techniques to vulnerability exploration and analysis. When facing massive input data sets, compared with traditional formal verification, symbolic execution, and fuzzing methods, machine learning has significant advantages in improving execution efficiency and reducing vulnerability detection costs. In addition, machine learning has the ability to self-learn, meaning that if a good model can be built, it can learn the features of smart contract vulnerabilities, thereby enabling smart contract vulnerability detection to free itself from the dependence on expert rules and manual operations [14].

However, current machine learning-based smart contract vulnerability detection techniques are still imperfect. Compared with other fields where machine learning is more mature, such as classification processing, natural language processing, etc., the primary challenge for smart contract vulnerability detection technology is how to preprocess smart contracts. Currently, most machine learning-based smart contract vulnerability detection methods treat smart contracts as text sequences, and then use Recurrent Neural Networks [15] and their variants in Natural Language Processing to extract vulnerability features. However, this causes a significant loss of syntax and semantic information in smart contracts, leading to unsatisfactory vulnerability detection results. Extracting the structural features in smart contracts and preserving the contract semantic information to the greatest extent is the key issue of vulnerability detection.

Smart contract source code, unlike the two-dimensional vector data of images and the serialized data of texts, exhibits a more complex structure. This structure demonstrates non-Euclidean characteristics in its spatial distribution [16]. Huang's research on vulnerability detection through bytecode matching [17] provides a new direction for our research: exploring the vectorization and graph representation of smart contract source code. More specifically, the source code is composed of basic elements such as contract declarations, state variables, functions, events, etc. These elements together form complex contract statements, with control and data associations among them. Thus, the representation of the source code involves preprocessing these elements and transforming them into data forms learnable by machine learning models, such as vectors and graphs. As a typical non-Euclidean data type, the graph includes nodes and edges as its fundamental components. The basic elements and some statements of smart contracts can be considered as nodes of the graph, while the relationships between these elements and statements form the edges. Therefore, one of our research goals is to develop an effective method for representing source code, aiming to preserve its semantic and structural features as much as possible, thereby providing a solid data foundation for learning and analyzing vulnerability characteristics.

In the field of smart contract vulnerability detection, selecting appropriate machine learning models is as crucial as the representation of code. Through an in-depth analysis of existing methods, we have identified the technical advantages and potential innovation space of Graph Neural Networks (GNN) [18] in this domain. When applied to the non-Euclidean space graphs generated by smart contracts, GNNs are capable of learning not only the local features of each node but also the global information between nodes, thereby more comprehensively revealing the syntactic and semantic features of smart contracts. Particularly, Graph Matching Networks (GMN) [19], a special form of GNNs, can maximize the capture and learning of the structural and semantic information of smart contracts through a cross-graph attention mechanism [19], offering a new solution to overcome the limitations in graph learning present in current research. Therefore, this thesis, focusing on smart contract vulnerability detection through code

representation and the GMN, holds significant practical importance.

Moreover, there are some difficulties in obtaining a large number of smart contract source codes that meet the experimental requirements. Obtaining real-world vulnerability data may include ethical issues. More specifically, since the source code of smart contracts often contains sensitive business logic and user data, such as transaction records, user identity information, and specific execution details of the contracts, using this data directly for model training could potentially violate data privacy protection regulations. It necessitates adopting stringent measures during data collection and processing, including but not limited to data anonymization and pseudonymization, to ensure the security and privacy of personal data are not compromised. For instance, this requires removing or replacing all personally identifiable information (e.g., addresses, transaction hashes) and employing encryption techniques to protect data integrity.

Acquiring private vulnerability data that includes ethical considerations is not straightforward, as it typically requires the consent of contract developers and may require complex legal and compliance processes. Moreover, processing such data requires specialized knowledge and skills to effectively identify and rectify potential vulnerabilities without infringing on the rights of data subjects. This process is time-consuming and costly, necessitating in-depth analysis by professional security analysts and data scientists. Therefore, utilizing open-source datasets that have been appropriately processed to remove sensitive information may be a more suitable choice. These datasets are often provided by community members, security researchers, or blockchain platforms and have been screened to ensure their legality and ethicality. By employing this approach, we can provide the necessary training data for smart contract vulnerability detection models while adhering to ethical and legal standards.

However, research has shown that only 1% of smart contracts are open-source [20]. 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 [21]. Additionally, manual data screening is necessary to ensure data quality and security. In the development of vulnerability detection models based on code representation and machine learning, special attention must be paid to the quality of training samples and the size of the training dataset. High-quality training samples are crucial for enhancing the accuracy and generalization ability of the model. Insufficient training data can lead to a decrease in model performance in terms of vulnerability detection. Therefore, inspired by data augmentation techniques in traditional machine learning [22], we aim to explore and implement few-shot learning approaches for smart contract vulnerability detection. When dealing with limited data availability, this method may offer an effective way to improve model performance and represents another key direction of our research.

1.3 Research Contributions and Overview

This thesis aims to investigate vulnerabilities in smart contracts and to develop vulnerability detection models by focusing on three key aspects: enhancing code representation, addressing the challenge of data scarcity, and applying advanced machine learning techniques.

Chapter 3 proposes a data processing model combining code embedding with Generative Adversarial Networks (GAN) [22]. This model significantly reduces the dependency on data quantity while enhancing vulnerability detection accuracy. Specifically, we initially employ code2vec [23] to obtain vector representations of smart contract source codes, and train the GAN model with vectors of contracts containing integer overflow vulnerabilities. Upon completion of training,

we utilize the GAN generator for data augmentation of the dataset and the GAN discriminator for preliminary vulnerability detection of target contracts, followed by a vector similarity analysis between the target and vulnerable contracts. In contrast to traditional methods, our approach delves into the Abstract Syntax Tree (AST) [24] of smart contract source codes and successfully achieves vectorization based on the AST, striving to preserve contract features to the utmost extent. The application of the GAN model not only generates the synthetic contract vector data but also enables the construction of detection models on small sample data, effectively mitigating the challenge of acquiring smart contract source codes. Moreover, our method combines feedback from the GAN discriminator with an integrated analysis of cosine similarity and correlation coefficients to implement dual similarity detection of contract vectors. This approach not only improves the accuracy of vulnerability detection but also has proven its effectiveness in practice. 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 develop a representation scheme for smart contract source code based on the AST, effectively preserving the contract's key features.
- 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.

Chapter 4 proposes a detection methodology that integrates GAN with GMNs [19]. This approach extensively leverages graph representation to capture the semantic and structural characteristics of smart contract code [17], while also utilizing GAN to address the shortage of training data. Given that traditional GANs are primarily suited for vector data enhancement, we opt for GraphGAN [25] for model development, which effectively learns graph representations and generates synthetic graph data. Specifically, we first extract the AST, Control Flow Graph (CFG), and Data Flow Graph (DFG) from the smart contract source code and merge them into a Contract Graph (CG) enriched with semantic and structural features. This multidimensional representation not only preserves the essential information of the contract but also intricately reflects the contract's control flow and data flow, surpassing traditional singular representation methods. By employing GraphGAN to augment small sample training sets, we enhance the diversity and quality of the training data, thus facilitating learning from small samples. Furthermore, we utilize GMN, equipped with a cross-graph attention mechanism [19], to learn key features of the CG, and assess the security of target contracts using the similarity scores generated by the GMN model. Empirical evidence demonstrates that this method effectively reduces the reliance on large data sets and ensures high precision and efficiency in the detection process. Our main contributions include the following points:

- We develop an innovative graph-based representation method for smart contract source code, which integrates AST, CFG, and DFG. This approach effectively preserves the structural and semantic features of the source code, providing a solid data foundation for the construction of the detection model.
- We use the GraphGAN model to generate a large number of synthetic CGs, achieving the goal of machine learning with a small-sample training set. This approach effectively addresses the issue of data starvation caused by the difficulty in obtaining smart contract

source code.

- We explore the graph feature learning capabilities of the GMN in smart contract vulnerability detection and achieve effective judgment of contract features.
- We innovatively propose a smart contract vulnerability detection method that combines GAN and GMN, which can effectively identify reentrancy and integer overflow vulnerabilities.
- We conducted experiments on 500 public Ethereum smart contracts, and the results show that our method has high accuracy, efficiency and scalability.

1.4 Thesis Structure

Chapter 1 introduces the research background and significance of this thesis. It first outlines the related concepts of smart contracts and provides a brief overview of the current application environment and security challenges of Ethereum smart contracts. It also summarizes the main technical paths for smart contract vulnerability detection at the present stage, and outlines our research motivation based on the current state of research. Then, we introduce our research content and results, and outline the structure of the thesis.

Chapter 2 provides a detailed introduction to the concepts and principles of blockchain, smart contracts, and Ethereum. It comprehensively summarizes and analyzes various influential types of vulnerabilities. It also compares the effectiveness of mainstream vulnerability detection techniques, highlighting the key features of their respective tools. This chapter not only introduces the representation of smart contract source code, but also provides the foundation for the methodology of the subsequent chapters. To be specific, this chapter elaborates on the GAN model under the few-shot learning objective. It explores the GMN in GNNs, laying the theoretical foundation for the contract similarity analysis in Chapter 4.

Chapter 3 proposes a smart contract vulnerability detection model based on code embedding and GAN. The model uses the GAN to generate a large amount of synthetic contract vector data, which preserves the structural and semantic features of real contracts. The model combines the feedback of the GAN discriminator and the code vector similarity analysis to detect integer overflow vulnerabilities efficiently, effectively alleviating the data starvation problem. Experimental results demonstrate the feasibility and effectiveness of the model in detecting integer overflow vulnerabilities in smart contracts.

Chapter 4 proposes a smart contract vulnerability detection method based on GAN and GMN, targeting the Ethereum platform's reentrancy and integer overflow vulnerabilities. The method uses GAN to solve the data starvation problem in training the GMN model. Specifically, the method transforms smart contract Solidity code into graphs containing semantic and structural features. Then, it uses graph representation-based GAN to augment the small-sample training set into a large-sample training set. The augmented training set is used to train GMN, which is an extended model based on GNNs. The model uses a cross-graph attention mechanism to calculate the feature similarity between the target and vulnerable contract. The experimental results show that the model has high accuracy and efficiency in detecting reentrancy and integer overflow vulnerabilities.

Chapter 5 summarizes the outcomes of this thesis, analyzes our research work, points out the problems existing in the vulnerability detection method we proposed, and looks forward to

future work.

Chapter 6 lists the publications of the author.

As shown in Figure 1.3, the inclusion of a diagrammatic representation of the thesis structure aims to clarify its organization and enhance the reader's comprehension of the flow.

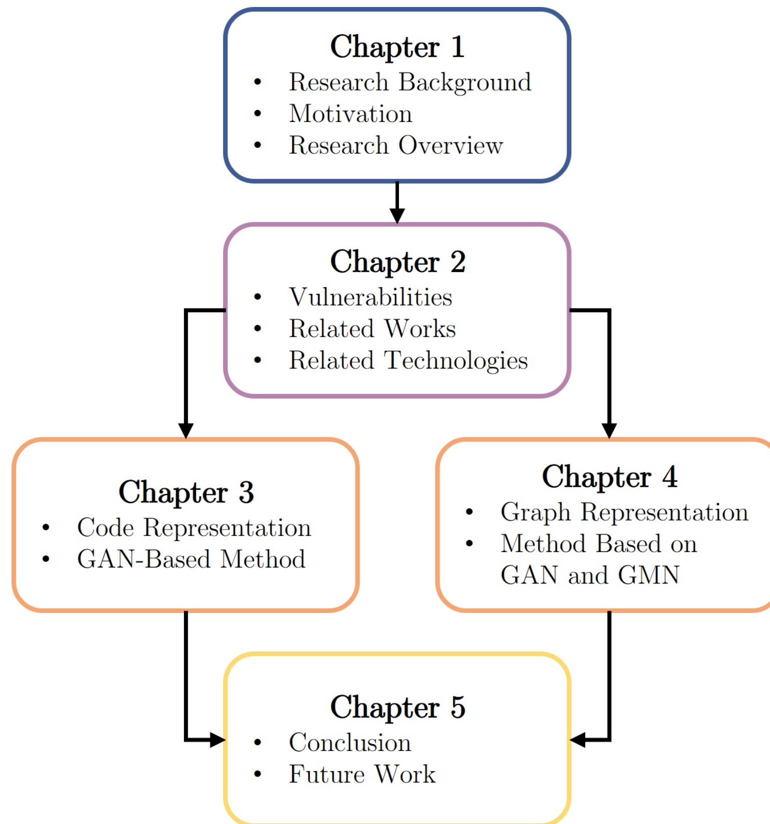


Figure 1.3: Diagram of the Thesis Structure.

Chapter 2

Literature Review

2.1 Introduction

Computer scientist and encryption expert Szabo first defined smart contracts as a kind of digital promise [1]. In 2008, a cryptographer named Nakamoto [2] created a virtual currency system called "Bitcoin" and formally proposed the blockchain concept. The blockchain technology that the system relied on was officially launched in 2009. Nakamoto believed that blockchain is a special data structure that forms a decentralized, shared, tamper-proof, and forgery-proof distributed ledger. Buterin [3] introduced smart contracts into blockchain technology in his published Ethereum whitepaper, which enabled blockchain applications to extend beyond the currency field to various other scenarios [26]–[28], thus ushering in a new era of blockchain 2.0.

Smart contracts are self-executing agreements running on blockchain platforms, such as Ethereum, the world's first and most prominent one. Smart contracts have various applications in fields like finance, gaming, insurance, education, and the Internet of Things (IoT). However, they also face different security threats and vulnerabilities throughout their life cycle, from creation to execution [29]. For instance, in the 2016 Decentralized Autonomous Organization (DAO) attack, the attacker exploited a split function vulnerability to withdraw multiple tokens and transfer them from the DAO main chain to a sub-chain, resulting in a loss of more than 3.6 million Ether assets [30]; in 2017, a bug in the multi-signature contract caused 150,037 Ether to be stolen from the Parity wallet; in 2018, the Beauty Chain project, which was worth tens of billions of dollars, collapsed to zero. A research survey indicates that 3.4% of Ethereum smart contracts contain varying vulnerabilities, involving approximately 4.4 million US dollars worth of Ether [10]. The details of smart contract vulnerabilities will be discussed in Section 2.3.

Smart contracts are vulnerable to various attacks that can compromise their execution and credibility. Therefore, developing effective and efficient detection schemes for smart contract vulnerabilities is crucial. Automated vulnerability mining [31] is a key research area in software security, which employs techniques such as formal verification [32], symbolic execution [33], fuzzing [34], program analysis [35] and taint analysis [36]. However, smart contracts are different from traditional programs in many aspects, such as the operating environment, life cycle, and program features, which bring new challenges to the automated vulnerability mining of smart contracts [31]. Many existing research works aim to adapt and apply existing techniques to smart contracts to achieve better vulnerability mining results. This chapter summarizes the current representative research works and vulnerability mining tools, which will be presented in Section 2.4. Moreover, we will also introduce and briefly describe a promising technique -

Deep Learning [37].

To explore new methods for detecting smart contract vulnerabilities, we will focus on the promising field of machine learning. As a representation learning technique, machine learning requires transforming the code into an appropriate form for model input when applied to code vulnerability detection. Therefore, the primary task in introducing machine learning to smart contract vulnerability detection is to encode the smart contract code into an appropriate representation. In section 2.5.1, we will discuss in detail the granularity and hierarchy of code representation, as well as the representation models involved in subsequent chapters. This will help us understand better how to effectively represent smart contract code for vulnerability detection, without compromising precision and accuracy.

Obtaining smart contract code that meets the requirements of machine learning poses several challenges, such as the scarcity of open-source contracts, privacy and legal issues, time and resource costs, and manual screening demands. These issues may lead to insufficient experimental data, which in turn reduces the accuracy and generalization ability of the detection models. Therefore, our research explores how to use a small amount of labeled data and a large amount of unlabeled data to improve the generalization ability and robustness of machine learning models, to achieve the goal of few-shot learning in smart contract vulnerability detection. In Section 2.5.2, we study and analyze Generative Adversarial Network (GAN) techniques in depth.

In our investigation and research of related work, we discovered that Graph Neural Networks (GNN) [38] are a type of deep learning model capable of handling graph-structured data. They can effectively extract features from nodes and edges within a graph and perform tasks such as node classification, graph classification, and link prediction. GNNs, as a connectionist model, capture the relationships within a graph through messages passing between nodes. One distinguishing feature of GNNs is their ability to aggregate information from neighboring nodes at arbitrary depths, setting them apart from standard neural networks. As an innovative model within the realm of GNNs, Graph Matching Network (GMN) [19] not only leverages information from node and edge attribute vectors but also harnesses structural information inherent in the data and control dependencies within smart contract source code. GMN utilizes this information to learn the vulnerability characteristics of smart contract source code and, consequently, to detect specific vulnerabilities present in the smart contract source code. We will provide a detailed explanation and analysis of the GMN in Section 2.5.2 to enhance understanding of its application in Chapter 4.

This chapter is divided into seven sections. Section 2.1 outlines the research topics covered in this chapter. Section 2.2 introduces the relevant concepts and applications of blockchain, Ethereum, and smart contracts. Section 2.3 summarizes and analyzes the vulnerability types with a wide impact and high severity. Section 2.4 analyzes and compares the mainstream vulnerability detection methods and representative tools. Section 2.5.1 introduces the representation of smart contract code, which lays the foundation for the methodologies in Chapter 3 and Chapter 4. Section 2.5.2 introduces machine learning techniques used in our research. To be specific, section 2.5.2 explains in detail the GAN that can achieve the goal of few-shot learning; section 2.5.2 introduces the Graph Matching Network in GNNs, which provides the theoretical basis for contract similarity analysis in Chapter 4. Finally, Section 2.6 concludes this literature review.

2.2 Research Background

2.2.1 Blockchain

Blockchain [39] is a distributed data storage technology based on cryptography, capable of achieving decentralized data management and transactions. From a narrow perspective, blockchain is an immutable distributed ledger composed of a series of data blocks connected in chronological order, with each block recording some transaction information and ensuring its security and consistency through cryptographic methods. From a broad perspective, blockchain is an innovative distributed computing model that uses a chain-like data structure to store data, distributed algorithms to transmit data, smart contracts to operate data, and consensus mechanisms to achieve digital consensus. Therefore, whether from a narrow or broad perspective, blockchain involves several core elements: distributed data storage, distributed ledgers, cryptographic encryption algorithms, smart contracts, and consensus mechanisms.

The data structure of blockchain is a special type of ledger that can contain digital transactions, data records, and executable files. These contents are divided into several larger units called blocks, each with a timestamp and a cryptographic link pointing to the previous block, forming an orderly record chain known as the "blockchain." This data structure not only describes the organizational method of the blockchain itself but also provides the foundation for building digital consensus architectures, algorithms, or application domains on top of it.

The data storage and transmission of blockchain are distributed, meaning that data is not centralized in a single central node or server but dispersed across multiple independent nodes, each holding identical or partially identical data copies. These nodes communicate and collaborate through a peer-to-peer network without relying on any centralized institutions or mechanisms. The consensus mechanism of blockchain is a node-voting-based method that can validate and confirm transactions within a limited time, ensuring a consistent state among all nodes. The encryption algorithm of blockchain comprehensively utilizes hash algorithms [40], symmetric encryption techniques, and asymmetric encryption techniques to ensure the non-falsifiability, integrity, and consistency of data on the blockchain. The smart contract technology of blockchain uses automated scripts to operate data, enabling various complex business logic and application scenarios.

A typical blockchain system usually consists of six layers [41]:

- Data layer: This layer is responsible for storing the data in the blockchain system, including the underlying data blocks composed of hash functions, digital signatures, timestamps, etc., and the blockchain structure formed by linking data blocks. The data layer ensures the integrity, immutability, and traceability of the data in the blockchain system.

- Network layer: This layer is responsible for implementing the communication and collaboration among nodes in the blockchain system, including establishing a distributed network using P2P protocol, propagating and verifying data using broadcast or sharding mechanisms, and solving malicious or faulty nodes in the network using Byzantine fault tolerance algorithm. The network layer ensures the data's consistency, reliability, and security in the blockchain system.

- Consensus layer: This layer is responsible for formulating the rules and mechanisms for nodes to reach consensus in the blockchain system, including selecting appropriate consensus algorithms, setting corresponding parameters and incentives, and determining

the role and power distribution in the consensus process. The consensus layer ensures the data's validity, fairness, and stability in the blockchain system.

- Incentive layer: This layer is responsible for designing the motivation and reward-punishment mechanism for nodes to participate in consensus and maintain the network in the blockchain system, including issuing tokens or points as incentives, formulating the distribution and circulation rules of tokens or points, and setting corresponding economic models and strategies. The incentive layer ensures nodes' cooperativeness, selfishness, and competitiveness in the blockchain system.
- Contract layer: This layer provides the programming and operation capabilities of data and logic in the blockchain system, including developing various scripting languages, algorithms, and smart contracts, and implementing the mapping and triggering relationships between data and logic. The contract layer ensures the blockchain system's flexibility, intelligence, and diversity of data.
- Application layer: This layer is responsible for developing various application scenarios and functions in the blockchain system, including using blockchain technology to realize digital currency transactions, decentralized applications (Dapp), enterprise-level blockchain applications, etc., and providing corresponding user interfaces and interaction modes. The application layer ensures the blockchain system's availability, usability, and value of data.

These six layers successively implement the basic functions of the blockchain system, network communication, transaction confirmation, system incentives, business logic, and application services.

2.2.2 Ethereum

Ethereum Platform

Blockchain technology has been rapidly developed and widely applied [42], giving rise to various blockchain platforms to meet the needs of different domains and scenarios. There are some of the current mainstream blockchain platforms, including Ethereum [43], Hyperledger [44], EOS [45], Cardano [46], TRON [47], Steller [48], NEO [49], Tezos [50], etc. There are multiple types of blockchain platforms, including public chains, private chains, and consortium chains. Public chains are open networks that anyone can participate in and access; private chains are closed networks that only specific organizations or individuals can participate in and access; consortium chains are semi-open networks that are jointly participated in and managed by multiple organizations or institutions.

We will focus on introducing and studying Ethereum, a highly popular and active public chain platform, which is also recognized as a key application of blockchain 2.0. Ethereum was created by Buterin in 2013. It is an open-source distributed computing platform that uses blockchain technology to implement a decentralized, transparent, secure, and reliable computing environment. It provides a complete development environment that allows users to develop and deploy smart contracts [3] easily. Smart contracts are special computer programs that can automatically complete specified tasks, but their execution process is irreversible. Once executed, they cannot be revoked or changed. This feature makes smart contracts very popular on the Ethereum blockchain and can be used to execute many different types of transactions and contracts.

The Ethereum platform consists of three main components: Ethereum Virtual Machine (EVM), execution environment, and consensus algorithm. EVM is a virtual machine based on Turing-complete scripting language for building applications. It can execute code of any complexity and has high flexibility. The execution environment refers to the part of the Ethereum network responsible for managing account states, transaction verification, Merkle trees, etc. The consensus algorithm refers to the Ethereum network mechanism responsible for achieving a consistent state among distributed nodes. Ethereum has transitioned to the Proof of Stake (PoS) [51] consensus mechanism, replacing the previous Proof of Work (PoW) [51] algorithm. This significant upgrade, known as "The Merge", was completed in 2022, and has reduced the blockchain's energy requirements by approximately 99.9% [52].

The Ethereum platform has the following advantages: First, it provides a comprehensive set of development tools, including compilers for building smart contracts with high-level languages such as Solidity, Vyper [53] and Yul [54], integrated development environments (IDEs), testing frameworks, debugging tools, etc., making it convenient for developers to develop and deploy their own applications. Second, it has a huge community and ecosystem, including developers, users, researchers, enterprises, organizations, etc., who provide rich resources, support, and innovation for the Ethereum platform. Third, it supports various types of applications, including crypto-collectibles, crypto-games, blockchain finance, proof-of-existence, etc., demonstrating blockchain technology's wide application scenarios and potential.

Ethereum Virtual Machine

EVM is the core component of the Ethereum platform. It is responsible for interpreting and executing the bytecode of smart contracts and maintaining the persistent storage associated with the blockchain state.

The design goal of EVM is to implement a universal, programmable, secure, and efficient distributed computing platform that can support various types of smart contract applications with different complexities. To adapt to the characteristics of cryptography and blockchain, EVM's data structure and instruction set are carefully designed. EVM's data unit is a 256-bit word, which matches the length of cryptographic techniques such as Keccak-256 hash or secp256k1 signature. EVM's instruction set includes standard arithmetic, logic, control flow, stack operations, and some blockchain-specific operations, such as obtaining addresses, balances, block hashes, etc.

The execution process of EVM can be divided into the following steps:

1) First, EVM receives an initial blockchain state and a set of valid transactions as input.
2) Second, EVM processes these transactions individually in order and updates the blockchain state and machine state according to the data and gas fees contained in the transactions.
3) If a transaction involves a contract account, EVM will execute the corresponding bytecode instructions according to the contract code and input data and may generate new messages.
4) EVM will recursively process these messages until all transactions and messages are processed, or gas fees are exhausted.
5) Finally, EVM will output a new blockchain state as the execution result of this group of transactions.

The advantage of EVM is that it provides a flexible and powerful programming model that can support various complex decentralized applications such as finance, social networking, games, etc. EVM also has high compatibility and portability. It can be implemented in different programming languages and platforms and integrated with various tools and frameworks. The security of EVM mainly depends on the protection of cryptography and blockchain as well as the quality and correctness of contract code. The efficiency of EVM depends on its implementation optimization and improvement as well as contract code design and optimization.

2.2.3 Smart Contract

Smart contracts are electronic data-based protocols that can automatically execute transactions according to predefined conditions. Szabo first proposed the concept of smart contracts in the 1990s. He believed such protocols could use computer programs to implement complex transactions without relying on third-party trust [1]. However, at that time, there was a lack of a suitable execution environment to ensure the trustworthiness and security of smart contracts, so smart contracts did not receive widespread application. In 2008, Nakamoto proposed blockchain as an innovative technology in his paper, which can realize a decentralized, immutable, and transparent distributed ledger [2]. Inspired by this, Buterin designed Ethereum as a new blockchain platform in 2014 and introduced smart contracts into it [3]. This made smart contracts not only applicable to digital currency transactions, but also to programming and managing the underlying data of blockchain, thus expanding the application scope and types of blockchain.

Smart contracts are representative technologies of the blockchain 2.0 era. Compared with Ethereum, the Bitcoin blockchain system [55] only has simple scripting functions and cannot execute Turing-complete programming languages. Ethereum smart contracts use Turing-complete programming languages for development and run in the EVM, which gives them higher flexibility and scalability.

In the architecture design of smart contracts, the following principles are generally followed: concise language, strong modularity, access control, and upgradability. Concise language means using high-level programming languages to write smart contracts, which can reduce the burden of developers on language details and avoid security vulnerabilities caused by language complexity. Strong modularity means dividing smart contracts into different parts, such as state variables, function interfaces, external receiving interfaces, callback processing, etc., which can improve the readability and maintainability of smart contracts. Access control means restricting the data flow in smart contracts to prevent unexpected operations. Upgradability means providing update and repair functions for smart contracts to increase functionality or fix bugs. In short, smart contract technology focuses on security and stability, ensuring that it is not maliciously interfered with on the blockchain platform.

The workflow of smart contracts can be summarized into three steps:

1) Construction: Users will obtain a pair of public and private keys after joining the blockchain network. The public key is the user's account address provided by the user; the private key is used to authenticate the user's identity and sign transactions or smart contracts, ensuring their security. Users negotiate and formulate an electronic protocol written in a programming language with other network participants and sign it with their private keys to ensure the contract's validity. Then, users upload the signed smart contract to the blockchain.

2) Storage: Every verification node on the blockchain will receive a copy of the smart contract. These contracts are propagated in a P2P manner in the network and stored on each blockchain node, forming a contract collection. In the next consensus round, nodes will process and verify these contracts. This process mainly involves hashing the contract collection and organizing the hash values into a block that is distributed throughout the network.

3) Execution: Smart contracts periodically check the state machines, transactions, and trigger conditions in the contracts. When the trigger conditions are met, transactions are automatically executed on the blockchain and marked by state machines. The execution results, such as changes in state, are recorded on the blockchain. The original contracts and their execution records are perpetually retained on the blockchain, as the data is immutable.

Application

Over time, smart contracts have become a popular choice in various fields, showing a wide range of application prospects in finance, management, healthcare, and the IoTs:

- In the financial field, smart contracts are naturally applicable. They can realize peer-to-peer transactions, greatly simplifying the transaction process without relying on strong third-party institutions. This helps improve transaction efficiency, reduce transaction costs, and provide more convenient tracing and investigation means for financial regulators, further ensuring investors' financial security. For example, Uniswap [56] is a decentralized exchange (DEX) allowing users to swap any ERC20 tokens without intermediaries or fees. MakerDAO [57] is a lending platform that allows users to borrow stablecoins (DAI) by locking up their collateral (ETH) in smart contracts. Augur [58] is a prediction market that allows users to create and bet on any event outcomes using smart contracts.

- In the management field, smart contracts have great potential. They can be combined with election and voting activities to ensure the fairness and transparency of voting results. In addition, smart contracts can also be used for digital asset copyright management, providing effective methods to track and protect intellectual property rights, while improving efficiency and accuracy in business process management. For example, uPort [59] is a self-sovereign identity (SSI) platform allowing users to create and manage their identities on Ethereum using smart contracts. Users can also use uPort to share their verifiable credentials with other parties in a secure and privacy-preserving way.

- In the healthcare field, smart contracts are also actively explored. One of the key applications is electronic medical record management. Smart contracts enable the setting of strict access permissions for electronic medical records, ensuring that only authorized personnel have access to the patient's treatment information, thereby effectively preventing the leakage and tampering of patient privacy. For example, MedRec [60] is a prototype system that uses Ethereum smart contracts to store and manage electronic medical records in a decentralized way. MedRec also provides incentives for network participants to maintain and verify data, as well as to enable data access and consent management for patients and providers.

- In the IoT field, smart contracts are closely integrated with the complex process of IoT [61]. They provide a solution that enables IoT devices to automatically execute

tasks and share resources, thus enhancing the industry's efficiency and information security while reducing application costs. For example, IOTA [62] is a distributed ledger technology that uses a Directed Acyclic Graph instead of a blockchain to achieve scalability and feeless transactions for IoT applications. IOTA also supports smart contracts that can run on IoT devices and enable various use cases such as data monetization, supply chain tracking, smart city, etc.

Development and Operation

Ethereum supports multiple smart contract languages, such as Solidity [63], Flint [64], Obsidian [65] and Lolisa [66]. Among them, Solidity is a high-level programming language similar to JavaScript, which is currently the most widely used smart contract language. The design goal of Solidity is to make smart contract code run effectively on EVM. In addition, Solidity also introduces an Application Binary Interface (ABI), which can implement multiple type-safe functions in a single smart contract. This thesis will take Solidity-written smart contracts as research objects.

To deploy and run smart contracts on Ethereum, developers need to go through the following steps:

1) Start an Ethereum node.

2) Use a smart contract language, such as Solidity, to write smart contracts.

3) Use the compiler to convert smart contract code into EVM bytecode.

4) Deploy the compiled smart contract code to the Ethereum network.

5) Use the JavaScript API interface provided by the web3.js library to call smart contracts.

2.3 Smart Contract Vulnerabilities

It is well known that software vulnerabilities are one or more defects in a program that malicious users can exploit to access or modify program data, interrupt normal execution, or perform unauthorized operations. Smart contract vulnerabilities are a special type of software vulnerability that can lead to resource locking or theft, identity tampering, compromised data integrity, and abnormal state changes during execution, thereby adversely affecting the interests of the users involved. Ethereum is currently the most influential open-source blockchain platform, as well as the platform with the most smart contracts, the most types of vulnerabilities, and the largest losses due to vulnerabilities. In addition, Ethereum smart contracts involve many typical vulnerability cases, and many smart contracts on other blockchain platforms are designed and written based on Ethereum smart contracts. Therefore, developers and researchers usually use Ethereum smart contracts as research objects for experiments and analysis. Therefore, this thesis will focus on Ethereum smart contracts.

As shown in Table 2.1, our research divides Ethereum smart contract vulnerabilities into three levels: Code Level, EVM Level and Blockchain Level. This section will describe and analyze the smart contract vulnerabilities at each level in detail.

Table 2.1: Classification of Smart Contract Vulnerabilities.

Level	Vulnerability Name	Cause of Vulnerability	Attack
Code Level	Integer Overflow	Improper Validation	BEC Token Attack
	Reentrancy	External Dependence	DAO Attack
	delegatecall Attack	Improper Function Use	Parity Multi-sig Wallet Hack
	Denial-of-Service	Improper Validation	King of the Ether Hack
EVM Level	Ether Lost in Transfer	Missing Proof	-
	tx.origin Attack	Improper Validation	THORChain Hack
	Short Address Attack	Missing Input Check	ERC20 Attack
Blockchain Level	Timestamp Dependency	Flexible Block Creation	-
	Transaction Ordering Dependency	Flexible Block Creation	-
	Block Information Dependency	Insecure Random Number	-

2.3.1 Code Level

Smart contracts written in Solidity, a programming language for smart contracts, may contain vulnerabilities due to the language's design defects or the developers' improper coding practices. These vulnerabilities can compromise the functionality and security of the contracts. We elaborate on and analyze four common vulnerabilities: Integer Overflow/Underflow, Reentrancy, *delegatecall* Attack, and Denial-of-Service.

Integer Overflow/Underflow

Integer overflow and underflow are common smart contract vulnerabilities that occur when the calculation result of an integer variable exceeds the representation range of its data type [17]. Overflow means that the calculation result is larger than the maximum value of the data type, causing the result to be truncated to a smaller value or zero; underflow means that the calculation result is smaller than the minimum value of the data type, causing the result to be truncated to a larger value. An integer overflow may cause logical errors or security vulnerabilities in smart contracts, such as attackers exploiting overflow vulnerabilities to steal funds or maliciously transfer funds. For example, in 2018, BEC tokens were attacked by hackers due to an integer overflow vulnerability, causing the token value to drop to zero.

Reentrancy

Similar to most programming languages, Ethereum smart contracts perform cross-contract function calls when processing business logic. However, smart contracts often involve sensitive operations such as transfers. A reentrancy vulnerability occurs when a contract does not properly update its state variables before an external call, allowing the target of the external call to re-enter the contract's internals and execute unexpected operations [67]. The cause of the reentrancy vulnerability is the $fallback$ mechanism of Solidity smart contracts. When a contract receives a transfer, it automatically triggers the $fallback$ function. If the $fallback$ function contains a call to the source contract of the transfer, it may form a recursive call, thereby affecting the normal logic of the target contract of the transfer. The reentrancy vulnerability is quite harmful, as it may cause assets in the contract to be stolen or locked. For example, an attacker designs malicious attack code in their callback function and recursively calls the victim contract's transfer function to steal Ether. The most famous reentrancy vulnerability incident was the DAO attack in 2016, where hackers exploited a reentrancy vulnerability and stole nearly 60 million dollars worth of Ether, forcing Ethereum to undergo a hard fork [30].

Delegatecall Attack

delegatecall is a special message call function. After the execution of a normal message call, the execution environment shifts, allowing the target code to run within the callee contract's context. To call a function of another contract, we need the ABI of the target function. If the ABI is known, then the signature of the target function can be used directly for calling purposes. But if we do not know the ABI of the target function, then we can use *delegatecall*. The peculiarity of *delegatecall* is that the execution environment does not change and the target code executes in the context of the caller contract, and the values of *msg.sender* and *msg.value* are still those of the caller, and the call stack remains unchanged as well.

The abuse of the *delegatecall* function can lead to serious security issues [68]. To be specific, the *delegatecall* called contract will directly modify the state variables of the caller, resulting in unexpected consequences. For example, in the famous Parity Multi-sig Wallet incident, the unrestricted *delegatecall* in the contract could call any Wallet Library smart contract function, and the wallet initialization function was not verified. The attacker used *delegatecall* to call the initialization function repeatedly, changed themselves to the contract owner, and transferred out the Ether in the contract.

Denial-of-Service

Deploying smart contracts or calling internal functions requires a certain amount of Gas to run. Gas is the computational resource in the Ethereum network, and each block has a Gas limit. If a transaction or a function exceeds this limit, it will fail. The Denial-of-Service (DoS) attack is a type of attack against smart contracts, aiming to make the contract unable to provide normal service or make the user unable to use the contract normally [69]. There are three common forms of DoS attacks:

- Reverting the contract state by *Revert* operation. Suppose the state change of the contract depends on the return value of an external function, and this function always returns errors or exceptions. In that case, the contract will always revert to its original state and fail to complete its expected functionality.
- Exhausting Gas to make the contract unable to execute. If the attacker deliberately

creates some complex or loop operations that consume a lot of Gas, the contract will fail due to exceeding the block Gas limit or making the user pay high Gas fees.

- Controlling *Owner* account to stop contract service. If an *Owner* account controls the contract and is capable of opening or closing the contract, an attacker gaining access to or tampering with this account could freeze or terminate the contract service at will.

These attack methods will bring security risks to smart contracts and affect their normal operation and reputation. For example, in 2016, KotET contracts suffered from DoS attacks, resulting in users being unable to use or withdraw their Ether stored in the contracts.

2.3.2 Ethereum Virtual Machine Level

EVM level vulnerabilities are caused by logical errors or inconsistencies when executing smart contracts bytecode, which may affect the interaction and collaboration among multiple contracts. These vulnerabilities may be due to defects or flaws in the design or implementation of EVM, resulting in unexpected behaviors or outcomes when executing contract code. We describe and analyze three common vulnerabilities: Ether Lost in Transfer [67], *tx.origin* Attack [68] and Short Address Attack [69].

Ether Lost in Transfer

When transferring Ether with smart contracts, the contract address of the recipient must be specified, and the address must be canonical. If Ether is transferred to a null address, the Ether will be lost forever [67]. It should be noted that a null address is not an address that has no association with any other user or contract. In fact, a null address is a special address that is designed to receive Ether but does not allow the transfer of this Ether to other addresses. The main purpose of a null address is to destroy Ether, thereby reducing the total supply of the Ethereum network.

Tx.origin Attack

Smart contracts can use the *tx.origin* variable to obtain the account address of the caller. *tx.origin* can traverse the call stack and return the account address variable that initiated the call, which can be used to verify or authorize transactions. However, it also has security risks [68]. If a contract uses the *tx.origin* variable to determine the user's identity, then an attacker can exploit this feature to steal ether from the contract. For example, an attacker can call the victim contract's withdrawal function in their own *fallback* function, inducing the victim contract to transfer ether to the attacker contract. Since the victim contract thinks that "*tx.origin* == *owner*", it cannot detect the anomaly, resulting in the ether in the contract being taken away by the attacker contract.

Short Address Attack

Smart contracts can use the EVM to process transaction messages. The input of transaction messages is composed of hexadecimal bytecode, which contains information such as method name hash, transaction target address, and transfer amount. This information has certain specifications in length, such as the contract address length must be 20 bytes. If the input contract address length is less than 20 bytes, the EVM will automatically fill "0" at the end to meet the requirements. However, this feature also provides an opportunity for attackers [69].

Attackers can deliberately use addresses with “0” at the end to call contracts, and then remove the “0” at the end of the address in the input parameters, so that the EVM takes out the missing bytes from the next parameter (such as ether quantity) to complete the address when parsing, and then fill “0” at the end of the entire input parameter. This will cause the parameter shift to expand, making the ether quantity left-shifted by one byte, resulting in the contract transferring excess ether to the attacker. This type of attack is called the Short Address Attack, which mainly targets token contracts based on the ERC20 type and will affect the normal operation and trustworthiness of smart contracts.

2.3.3 Blockchain Level

Blockchain level vulnerabilities are caused by design flaws or implementation errors of the blockchain system itself, which may affect the stability and security of the entire blockchain network. For example, Timestamp Dependency [70], Transaction Ordering Dependency [68], and Block Information Dependency [69]. These vulnerabilities may be due to the lack of sufficient consideration of potential attack scenarios or threat models when designing or deploying the blockchain system, resulting in security risks or weak links in the blockchain system during operation. We will describe and analyze the above three common vulnerabilities.

Timestamp Dependency

The timestamp on the blockchain is an important mechanism that provides order and immutability for the blocks and transactions. The timestamp is a specific hash value generated by the miner (the node or user in the blockchain network) who creates a new block, which reflects the time when the block is generated, and cannot be modified by anyone. Smart contracts can utilize the block timestamp (*block.timestamp*) to enforce time constraints, enabling certain operations to be executed only within designated timeframes. The block timestamp also ensures the consistency of the state after the execution of smart contracts, because all transactions in the block share the same timestamp. However, the block timestamp entails certain risks, as miners can manipulate it to a degree, potentially causing deviations from the actual time. If smart contracts do not correctly handle timestamp dependency, attackers may exploit this to bypass the restrictions of smart contracts or achieve other malicious purposes [70].

Transaction Ordering Dependency

Transaction ordering dependency is a type of attack that mainly targets signatures [68]. This vulnerability occurs when the state change or logic judgment of the contract depends on the execution order of transactions in the block, which is determined by the miner, rather than controlled by the contract itself. This vulnerability may compromise the functionality or security of the contract. The reason for transaction ordering dependency vulnerability is that the transaction signatures on the blockchain do not contain all the hash encryption information, so attackers can modify some parameters without changing the signature, to affect the ranking of transactions in the undetermined transaction pool. If attackers are able to observe transactions in the pool and collaborate with or become miners themselves, they can manipulate transaction orders within a block to execute attacks.

Block Information Dependency

Random numbers are a common mechanism in smart contracts, which can be used to implement scenarios based on random selection, such as lottery and draw. However, generating random numbers in Solidity is a difficult task, because Ethereum smart contracts cannot directly access true random sources. Therefore, contract developers usually write their own random functions, using block-related parameters or information such as block number (*block.number*), block timestamp (*block.timestamp*), or block hash (*block.blockhash*) as the seed of random numbers. However, this method poses a security risk as miners can foresee these block parameters, rendering the generated random numbers predictable. If there are malicious attackers who can monitor or manipulate these block parameters, they may use this to generate random numbers that are favorable to them.

2.4 Related Research on Vulnerability Detection

How to effectively detect and repair smart contract vulnerabilities, which seriously threaten the execution and credibility of smart contracts, is a hot issue in the field of blockchain security. Automated vulnerability mining is an important research area of software vulnerability mining [31], and existing works focus on applying existing methods in the field of software vulnerability detection to smart contracts. However, due to the significant differences between smart contracts and traditional software in terms of running environment, life cycle, and characteristics, this also brings new challenges and demands for the automated vulnerability mining of smart contracts. Table 2.2 presents a list of representative detection methods that have been analyzed in detail within this section, along with their respective publication years, citation counts as recorded in Google Scholar, and unique contributions. It should be noted that Mythril [71], an open-source tool, does not have a citation count listed.

This section will systematically review and analyze the methods of automated vulnerability mining of smart contracts, mainly including the following aspects:

- According to the technical characteristics, the traditional methods of vulnerability detection of smart contracts include several types: Formal Verification [9], Symbolic Execution [10], Fuzzing [11], etc. We will introduce these three methods' principles, features, and representative works, respectively.
- We explore the detection methods based on deep learning [72] and analyze its principles, features, and representative works.
- We summarize the limitations and shortcomings of existing methods and introduce our research motivation and design ideas.

2.4.1 Fuzzing

Fuzzing is a dynamic vulnerability detection technique that detects software vulnerabilities by generating a large number of normal and abnormal inputs for the program. The key to fuzzing is to generate different types of random inputs, which mainly have two ways of processing: generation-based and mutation-based. The generation-based method is suitable for software programs with strict input formats, and generates input cases according to a certain format; the mutation-based method modifies the initial input based on the feedback of the program

Table 2.2: Classification of Smart Contract Vulnerability Detection Methods.

Method Principle	**Method Name**	**Publication Year**	**Citation Count**	**Unique Contribution**
Fuzzing	sFuzz	2020	206	AFL-based Adaptive Strategies
	ContractFuzzer	2018	605	EVM Instrumentation ABI-Compliant
	ILF	2019	206	Neural Augmentation Symbolic Coverage
	IR-Fuzz	2023	25	Data-Driven Ordering Targeted Efficiency
Symbolic Execution	Oyente	2016	2304	EVM Simulation Z3 Analysis
	Maian	2018	628	Customized EVM Concrete Verification
	Mythril	2018	N/A	LASER Execution Z3 Proof
	Park	2022	11	Parallel Framework Adaptive Management
Formal Verification	Securify	2018	931	Datalog Analysis Compliance Detection
	KEVM	2018	415	Executable Semantics Extensibility
	ZEUS	2018	740	Static Analysis SeaHorn Verification
	FSM	2023	6	FSM Model Structured Process
Deep Learning	CGE	2021	155	Graph-Expert Integration Semantic Extraction
	Bi-GGNN	2023	13	Bidirectional Representation Hybrid Attention
	CodeNet	2022	23	CNN Analysis Image Detection
	BLSTM-ATT	2020	112	Bidirectional LSTM Attention Focus

execution process, thereby generating different input cases. Fuzzing is widely used in the traditional software vulnerability detection field, and its effectiveness has been verified [34].

Fuzzing has good scalability and usability in smart contract detection and can perform vulnerability mining without obtaining black boxes. Its execution process includes:

1) Preprocessing contracts, building Control Flow Graphs (CFG), and collecting function dependencies.
2) Generating input data according to random mutation and coverage-based feedback strategies.
3) Executing contracts and collecting coverage, exceptions, state changes, and other information generated during operation.
4) Identifying contract vulnerabilities and further optimizing test cases.

Some representative tools will be elaborated as follows.

sFuzz

sFuzz [11] is a gray-box fuzzing tool based on American Fuzzy Lop (AFL) [73], which is designed for uncovering vulnerabilities in smart contracts. The tool generates transaction inputs by parsing the bytecode of smart contracts and executes these inputs in a modified EVM. By analyzing the feedback provided by EVM, sFuzz can discover various vulnerabilities, such as reentrancy, integer overflow, and call stack overflow. This approach integrates the strengths of fuzzing and symbolic execution, enabling it to cover a broader range of code and detect vulnerabilities that are challenging to uncover through black-box fuzzing. Examples of such hard-to-find vulnerabilities include those in the DAO contract and the Parity Wallet contract [11]. In addition, sFuzz also adopts an adaptive seed selection strategy, dynamically generates CFGs, skips functions that do not affect the state, and introduces mutation strategies tailored for smart contracts.

sFuzz workflow primarily consists of the following steps:

1) Initialization: First, sFuzz generates an initial test suite based on the bytecode and ABI of the smart contract. This test suite includes a set of randomly generated test cases, each containing an initial configuration and a series of function calls.
2) Execution of test cases: Next, sFuzz executes these test cases and monitors the execution process to collect feedback. This feedback includes information on which branches are covered and the extent of the coverage.
3) Seed selection: Based on the collected feedback, sFuzz employs two strategies to select seeds. One is the AFL strategy, which chooses test cases that cover new branches. The other is the adaptive strategy, which selects seeds based on the distance between the test case and the uncovered branches.
4) Crossover and mutation: sFuzz uses genetic algorithms to perform crossover and mutation on the selected seeds, generating new test cases. These new test cases will be used in the next round of execution.
5) Iterative execution: sFuzz repeats the above steps until a preset time limit is reached. Throughout the process, sFuzz continuously optimizes the test suite to improve code coverage and discover potential vulnerabilities.

In summary, the sFuzz workflow employs adaptive fuzzing to generate and execute test cases to identify vulnerabilities in smart contracts and enhance code coverage. By combining the AFL and adaptive strategies, sFuzz achieves more efficient and effective testing.

ContractFuzzer

ContractFuzzer [74] is a fuzzing tool for Ethereum smart contract security vulnerability detection, mainly consisting of an EVM instrumentation tool and a fuzzing tool. The EVM instrumentation tool can instrument the EVM to monitor the execution process of smart contracts and extract execution logs for vulnerability analysis. The fuzzing tool can analyze the parameter types of smart contract functions based on the ABI of the smart contract and generate input test cases compliant with the ABI specification for fuzzing. ContractFuzzer can detect seven types of smart contract security vulnerabilities, including gasless send, exception disorder, reentrancy, timestamp dependency, block number dependency, dangerous *delegatecall*, and freezing ether.

The workflow of ContractFuzzer consists of six steps:

1) Offline detection of the EVM to enable the fuzzing tool to monitor the execution of smart contracts and extract vulnerability information. Simultaneously, a web crawler is used to fetch smart contract source code, binary files, ABIs, and constructor parameters from the Ethereum blockchain browser.
2) Analysis of the ABI and smart contract bytecode to extract parameter data types and function signatures.
3) Signature analysis and indexing of all smart contracts in the third step.
4) Generation of valid and mutation input test cases compliant with the ABI specification. The ABI can specify smart contract addresses and function selectors as parameters.
5) Transmission of inputs to the ABI through random function calls.
6) Analysis of execution logs generated during the fuzzing process to detect security vulnerabilities. Fuzzing will continue until all smart contracts have been tested.

ILF

ILF [75] is a method that combines neural network models and fuzzing to improve the coverage of fuzzing. This method uses a symbolic execution engine to generate random input and uses a deep learning network to learn the input features that can improve the coverage. It learns fast and effective fuzzes from symbolic execution under the imitation learning framework by learning task descriptions. Symbolic execution experts generate a large number of high-quality inputs to improve the coverage of thousands of programs. Subsequently, a fuzzing strategy represented by a suitable neural network structure is trained on the generated data set to perform fuzzing on smart contracts. This tool instantiates an approach for dealing with fuzzy smart contract problems and proposes an end-to-end detection system.

IR-Fuzz

IR-Fuzz [76] stands out as an innovative smart contract fuzzing tool that offers significant advantages over traditional fuzzing tools by introducing a novel approach to vulnerability detection in Ethereum contracts. Its unique sequence generation strategy intelligently orders function

calls based on data dependencies, allowing for a more in-depth exploration of the contract's state space. This contrasts with conventional tools that often generate function invocations randomly, potentially missing critical state transitions that could reveal vulnerabilities. Additionally, IR-Fuzz's seed optimization technique, which iteratively evolves test cases towards less reachable branches, addresses the limitations of traditional fuzzers that may struggle to find complex bugs due to their random nature. The energy allocation mechanism in IR-Fuzz further enhances its efficiency by directing fuzzing efforts towards branches that are rare or more likely to contain security issues, thus avoiding the resource wastage often seen in tools that treat all branches equally. These innovations collectively enable IR-Fuzz to achieve higher branch coverage and more accurate vulnerability detection, setting a new standard in smart contract security analysis.

In addition, some detection methods have been proposed in the recent two years. CrossFuzz [77] is a cross-contract vulnerability detection method that generates constructor parameters by tracing the data propagation paths of constructor arguments. It optimizes the mutation strategy of transaction sequences based on inter-contract data flow information, thereby enhancing the efficiency and vulnerability detection capabilities of fuzzing. The innovation of CrossFuzz lies in its use of data flow information to refine mutation strategies, significantly improving the efficiency of detecting cross-contract vulnerabilities.

ItyFuzz [78] is a snapshot-based fuzzer for smart contracts. It explores intriguing states and rapidly synthesizes concrete exploits, such as reentrancy attacks, by storing and mutating snapshots of smart contract states rather than transaction sequences. The novelty of ItyFuzz lies in its introduction of data flow and comparison waypoints to optimize state exploration and storage. Its support for on-chain testing enables it to discover and generate real vulnerabilities of on-chain projects quickly.

2.4.2 Symbolic Execution

Symbolic execution [10] is a traditional automated technique widely used for detecting vulnerabilities in smart contracts. This technique provides a symbolic virtual running environment for the target code, abstracts the external inputs required by the program into symbolic values with indefinite values, and continuously solves path constraints to explore program branches. The main idea is to convert uncertain inputs during program execution into symbolic values, thereby promoting program execution and analysis.

Both smart contract programs and traditional programs can be abstracted as an execution tree. In the normal execution process, since the program input is a fixed value, each conditional judgment can get a definite answer, so only one branch is explored. However, in symbolic execution, the input value is an indefinite symbolic variable. When encountering a conditional judgment, the symbolic execution engine will use a constraint solver to solve the expression containing the symbolic variable. For all branches with solutions, symbolic execution will analyze and record the constraints in the path.

This section will propose and analyze four tools that use symbolic execution methods. These tools provide an effective method for detecting potential vulnerabilities and security issues for smart contracts.

Oyente

Oyente [10] is one of the earliest research works that applied symbolic execution to smart contract vulnerability detection. Oyente can explore all executable paths of smart contracts by simulating EVM, and use Z3 solver to determine whether the paths are reachable and whether there are security issues. Oyente can discover various types of smart contract vulnerabilities, including timestamp dependency, reentrancy, integer overflow, unchecked return value, etc., and output the problematic symbolic paths and related information to users.

The architecture of Oyente consists of four parts: CFG constructor, resource manager, core analyzer, and verifier. The CFG constructor is responsible for generating the CFG according to the input smart contract bytecode, where each node represents a basic execution block, and each edge represents an execution jump. The resource manager is responsible for performing symbolic execution on the CFG and collecting information during the execution process, such as state variables, function calls, exception handling, etc. The core analyzer conducts security analysis on each path, utilizing the collected information and predefined vulnerability patterns. It employs the Z3 solver to address path conditions and detect vulnerabilities. The verifier is responsible for filtering and optimizing the output results of the core analyzer, removing some false positives or duplicate data.

Maian

Maian [79] is a symbolic analysis tool designed to identify smart contract vulnerabilities. It specifies and infers properties of execution traces and employs a concrete verifier to confirm the authenticity of the identified vulnerabilities. Maian consists of two parts: a symbolic analyzer and a concrete verifier. The symbolic analyzer constructs a customized EVM according to the input contract bytecode and analysis specification (including the type of vulnerability to be detected and the search depth), and performs symbolic execution on the contract bytecode, exploring all possible execution paths, and finding paths that satisfy the preset properties. Each path corresponds to an input context, which is a set of symbolic variables. The symbolic analyzer returns the concrete values of these variables as output. The concrete verifier receives the output of the symbolic analyzer and re-executes the contract with these values as input, checks whether there are vulnerabilities, and filters out some false positives or duplicate results. Maian can detect three types of smart contract vulnerabilities: Greedy, Prodigal, and Suicidal.

Mythril

Mythril [71] is a smart contract security analysis tool specifically designed for EVM bytecode, playing a significant role in the Ethereum smart contract domain. This tool aims to assist developers in identifying and resolving potential security vulnerabilities, thereby ensuring the security and reliability of smart contracts.

The working principle of Mythril is based on symbolic execution engines, which traverse all possible states of smart contracts through function calls. The tool utilizes LASER (a symbolic virtual machine) to create an environment for executing bytecode and revealing vulnerabilities. LASER is responsible for calculating all possible program states. Mythril also employs Z3 to prove or disprove the extensibility of states. Through this approach, Mythril can detect various types of security vulnerabilities, such as integer overflow, unprotected Ether extraction, and DoS attacks. Mythril associates the detected vulnerabilities with official vulnerability documentation, enabling users to understand and address these issues.

Park

Park [80], as a pioneering framework for smart contract vulnerability detection, introduces a paradigm of parallel-fork symbolic execution that harnesses the power of multiple CPU cores to dynamically fork processes, thereby amplifying its detection capabilities. It addresses the challenges of parallel SMT solver performance by incorporating an adaptive algorithm that dynamically manages the number of active processes and adjusts the solver's timeout settings to optimize path exploration. Park also innovates by utilizing a shared-memory approach to maintain and reconstruct global variables across parallel processes, which is essential for identifying vulnerabilities that require a holistic view of the contract's state.

The framework's operational sequence begins with inputting a smart contract into a symbolic execution environment, followed by state synchronization, process forking, and SMT timeout tuning. It continues with collecting global variables into shared memory and reconstructing these variables for comprehensive vulnerability analysis. Park's integration as a plugin with established symbolic execution tools like Oyente and Mythril enhances its versatility and ease of adoption.

Moreover, there are some noteworthy new methods. AChecker [81] employs symbolic execution analysis to infer access control state variables and critical instructions in contract code that could potentially be manipulated by unauthorized users, thereby distinguishing between intended behaviors and potential security vulnerabilities as envisioned by the contract developers. The innovation of this research lies in its combination of static data flow analysis and symbolic execution to detect access control vulnerabilities with high precision while reducing false positives.

EtherSolve [82] can accurately reconstruct the CFG from the EVM bytecode of Ethereum smart contracts. By symbolically executing the operand stack to resolve jump addresses, EtherSolve enhances the accuracy of CFG reconstruction, which is crucial for subsequent vulnerability detection. Its innovation lies in its ability to handle smart contracts compiled by various versions of Solidity, compute a detailed CFG in almost all cases, and detect vulnerabilities such as reentrancy and $tx.origin$.

2.4.3 Formal Verification

Formal verification [83] is a method that uses rigorous mathematical foundations and logical reasoning to analyze and prove the functionality and properties of smart contracts. The core idea of formal verification is to express smart contracts in a precise and verifiable description language or logic, then describe their properties according to the design specification, and automatically perform mathematical analysis and proof.

Formal verification can be divided into two main methods: model checking and deductive verification. Model checking is to verify whether a smart contract satisfies a given specification or property by constructing a finite state model of the smart contract and traversing all possible states. Deductive verification involves describing the smart contract and its properties using logical formulas, and then proving that the contract exhibits certain characteristics or meets specific conditions through various reasoning rules. Formal verification can effectively discover various types of vulnerabilities in smart contracts, thereby improving the credibility and reliability of smart contracts. This section will propose and analyze three tools that use formal verification.

Securify

Securify [9] is a smart contract security analysis tool based on formal verification, which can detect various types of vulnerabilities in contracts, such as reentrancy, timestamp dependency, integer overflow, unchecked return value, etc. The core idea of Securify is to convert the contract bytecode and semantic information into facts and rules of the Datalog language, and compare them with predefined security property rules to judge whether the contract satisfies or violates a certain security property. Securify's security property rules are divided into two modes: compliance mode and violation mode, which represent the correct or incorrect behavior of the contract.

Securify's workflow is as follows: First, Securify extracts the contract's dependency and control flow information from the input contract bytecode, which is done by the bytecode analyzer. Second, Securify obtains precise semantic information from the bytecode, such as state variables, function calls, exception handling, etc., which is done by the semantic information extractor. Then, Securify converts the semantic information and security property rules into the form of the Datalog language, and generates corresponding facts and rules, which is done by the Datalog converter. Finally, Securify matches and checks the semantic facts and security property rules, and outputs the detection results, which the pattern matcher does.

KEVM

KEVM [84] is an executable formal semantics of the EVM implemented using the K framework. The primary significance of KEVM lies in providing comprehensive, accurate, and executable semantics for Ethereum smart contracts, thereby establishing a foundation for the analysis, verification, and development of smart contracts. KEVM exhibits the following notable characteristics: As an executable semantics, KEVM enables the actual execution of Ethereum smart contracts. This allows developers to prototype and test smart contracts on KEVM, thereby identifying potential issues prior to actual deployment; KEVM's semantics are based on the Ethereum yellow paper [85], ensuring that it accurately reflects the behavior of the EVM. This makes KEVM a reliable tool for analyzing and verifying Ethereum smart contracts. Despite being an executable semantics, KEVM demonstrates excellent performance. In comparison to existing EVM semantics implementations, KEVM exhibits superior performance in numerous test cases; KEVM's semantics possess strong extensibility, allowing for the easy addition of new features and tools. For example, KEVM can be utilized for developing smart contract analysis tools, such as gas analysis tools and ABI-level DSLs.

ZEUS

ZEUS [86] is a static analysis tool based on formal verification, which can quickly verify the security of contracts by abstract interpretation, symbolic model checking, and constraint statements, and support the detection of various smart contract vulnerabilities, such as reentrancy, integer overflow, block state dependence, etc. The workflow of ZEUS is as follows: First, convert the Solidity code into a format acceptable by SeaHorn [87] using an intermediate language; Then, apply verification rules written in a specific language on the converted code; Finally, use SeaHorn to perform formal verification and output the detection results.

FSM-Based Verification

The FSM-based verification model [88] introduces a novel methodology for ensuring the security and correctness of composite smart contracts on the Ethereum blockchain. This model operates through a structured process that encompasses three key phases: modeling, formal specification, and verification. In the modeling phase, the model translates Solidity smart contracts into FSMs, providing a detailed behavioral representation of the contracts. The formal specification phase then defines security and correctness properties using CTL formulae, which are essential for capturing the contracts' intended behavior and security requirements. Finally, the verification phase utilizes the nuXmv symbolic model checker to rigorously test the FSM model against these properties, thereby identifying vulnerabilities and confirming the contracts' functionality.

The model's innovative aspect is its dual-tiered verification approach, which integrates both standard and context-specific security properties. This approach is particularly effective for composite smart contracts, as it addresses a broad range of vulnerabilities, from common security issues to those unique to individual contracts. The model's application to various Solidity smart contract examples has demonstrated its capability to enhance security assurance within the realm of blockchain-based collaborative environments.

2.4.4 Deep Learning

It is well known that deep learning techniques [37] in the field of artificial intelligence have achieved good results in fields such as computer vision, speech recognition, and natural language processing [89]. Facing the increasingly serious smart contract security problems and the limitations of traditional detection methods, more and more researchers are trying to apply deep learning to vulnerability detection schemes. Deep learning models, as a kind of multi-layer neural network model, can automatically extract and abstract high-level features of data, avoiding the tedious manual feature extraction process. The core idea of deep learning methods is to extract vulnerability-related features from smart contracts and use these features to train deep learning models, thereby achieving the detection of vulnerabilities in smart contracts. The structure of smart contract features determines the vulnerability-related information that deep learning models can learn from, so different feature extraction methods will affect the detection effect. In addition, the model's selection, construction, and training are also important factors affecting the detection effect. This section will describe and analyze three popular deep learning methods.

Graph Neural Network

GNN [90]–[92] is a deep learning model specially designed for processing graph-structured data. The core idea of GNN is to learn node representation and graph representation through the information of nodes and neighbor nodes. GNN usually consists of two main parts: node update and graph update. Node update refers to the process of updating node representation according to the node's own features and neighbor node features. Graph update refers to the process of updating graph representation according to node representation. Through these two processes, GNN can capture the complex relationship between nodes and learn the high-level representation of the graph.

Taking Combining Graph feature and Expert patterns (CGE) [18] as an example, this is a smart contract vulnerability detection method that uses GNNs and expert patterns, which can

efficiently detect vulnerabilities such as reentrancy, timestamp dependency, and infinite loops. The basic principle of CGE is to convert the smart contract source code into a graph and then use GNNs to learn the semantic features of the contract graph. These features are combined with the security characteristics of expert patterns to produce results for vulnerability detection.

CGE's workflow includes the following steps:

1) Extracting expert patterns related to vulnerabilities from the source code, including vulnerabilities' definition and detection methods, as guidance for subsequent detection.

2) Representing the source code as a contract graph, where nodes represent key variables and function calls, and edges represent control flow and data flow relationships. The graph reflects the smart contract's semantic information, which helps discover potential vulnerabilities.

3) Designing a node elimination stage to simplify the contract graph, highlighting important nodes. This step helps to eliminate redundant information and improve detection efficiency.

4) Using a temporal message propagation network to learn the semantic features of the contract graph automatically. This step leverages GNNs' ability to capture the complex relationships between nodes and thus learn a high-level representation of the graph, which helps to discover vulnerabilities in smart contracts.

5) Combining the graph features with the expert pattern features, and generates the final vulnerability detection results. This step combines the advantages of expert knowledge and GNNs to improve the accuracy and robustness of detection.

Unlike the CGE, the Bi-GGNN (Bidirectional Gated Graph Neural Network) model [93] offers an approach by integrating a sliced joint graph representation that amalgamates Abstract Syntax Tree (AST), CFG, and Program Dependence Graph to capture both syntactic and semantic features of smart contract functions. The model employs program slicing to eliminate redundant information, thereby focusing on vulnerability-relevant components and reducing noise interference. It further utilizes a bidirectional learning mechanism to capture contextual features from both precursor and successor nodes, which is crucial for accurately identifying vulnerabilities. The model's innovative hybrid attention pooling layer combines self-attention and global attention to construct a graph-level feature representation, enabling it to discern task-relevant nodes and suppress less significant ones. Empirical results demonstrate that the Bi-GGNN model achieves superior precision and recall rates compared to existing approaches, highlighting its effectiveness in smart contract vulnerability detection. The model's innovative aspects include its ability to learn from a rich code representation, bidirectional feature learning, and hybrid attention mechanism, all of which contribute to its enhanced detection capabilities.

The successful application of CGE and Bi-GGNN further demonstrates the advantages of GNNs in detecting smart contract vulnerabilities: GNNs are capable of processing large-scale graph data and are well-suited for scenarios with an increasing number of smart contracts; GNNs enable automatic learning of node and graph representations, reducing the manual feature design's necessity.

Convolutional Neural Network

Convolutional Neural Network (CNN) [94] is a deep learning model mainly used for processing data with grid structure, such as images and speech signals. CNN is built by components such

as a convolutional layer, activation function, pooling layer, and fully connected layer. The convolutional layer is used to extract local features, the activation function introduces non-linearity, the pooling layer is used to reduce data dimension, and the fully connected layer is used for the final classification task. CNN learns the high-level abstract representation of data through a multi-layer structure, thus achieving efficient processing of complex data.

CNN also has significant advantages in smart contract vulnerability detection. Taking CodeNet [95] as an example, this is a smart contract vulnerability detection method based on CNN. CodeNet designs a new CNN architecture to detect vulnerabilities while maintaining the semantics and context of smart contracts. In order to improve the performance of CodeNet, the authors also designed a data preprocessing method, which converts smart contracts into images while maintaining locality. CodeNet performs vulnerability detection through the following steps:

1) Data preprocessing: First, compile the smart contract source code into bytecode. Then, convert the bytecode into fixed-size code. To generate input images, map the fixed-size code to contract-based images. In the mapping process, the authors used different filter sizes to extract features of instruction sequences with different lengths.

2) Vulnerability detection: CodeNet analyzes contract images and identifies vulnerabilities. In order to perform vulnerability detection while maintaining semantics and context, CodeNet adopts the following strategies:

    - No stride: CodeNet does not use stride to avoid losing local information. This enables the model to capture semantic and contextual information that depends on pixels.

    - Using depthwise separable convolution: To reduce computation and parameter quantity, CodeNet uses depthwise separable convolution.

    - Using global max pooling: CodeNet uses a global max pooling layer instead of a global average pooling layer to preserve original information. The global max pooling layer can retain the maximum activation value in the feature map, thus avoiding losing important information.

Through these strategies, CodeNet can perform efficient and accurate vulnerability detection for reentrancy, unchecked low-level calls, timestamp Dependency, and *tx.origin*. CodeNet highlights the efficacy of CNNs in smart contract vulnerability detection, showcasing their ability to rapidly process large datasets for timely vulnerability identification, learn high-level abstract data representations for improved accuracy, and autonomously extract features from training data without relying on predefined rules.

Recurrent Neural Network

Recurrent Neural Network (RNN) [15] is a deep learning model mainly used for processing data with sequential structure. RNN combines the current input with the hidden state of the previous moment through recurrent connections, thereby capturing the dependency relationship in the sequence. However, traditional RNNs are prone to gradient vanishing or gradient explosion problems when dealing with long sequences. To solve this problem, variants of RNNs such as Long Short-Term Memory (LSTM) [96] and Gated Recurrent Unit (GRU) [97] have been proposed.

BLSTM-ATT [98] is a smart contract vulnerability detection method based on RNN. First, preprocess the smart contract source code, extract key information and construct input data. Then, design a Bidirectional LSTM (BLSTM) model and combine it with Attention Mechanism. BLSTM can capture the long-distance dependency relationship in the input sequence, and the attention mechanism helps the model focus on the key parts of the input data. Finally, optimize the model parameters and improve the detection performance through the training and verification process. BLSTM-ATT's detection process for smart contracts is as follows:

1) Data preprocessing: Clean up the smart contract source code, extract key information, and construct input data.
2) Model construction: Design the BLSTM-ATT model, including the LSTM layer, attention layer, and fully connected layer components.
3) Model training: Use the training data set to optimize the model parameters through the backpropagation algorithm.
4) Model verification: Use the verification data set to evaluate the model performance and adjust hyperparameters.
5) Vulnerability detection: Input the smart contract source code to be detected, and the model outputs the vulnerability type and location.

BLSTM-ATT demonstrates that RNN excels in smart contract vulnerability detection due to its robust feature extraction capabilities, particularly in extracting sequential features from input data. This ability is crucial for identifying vulnerabilities. Additionally, RNN effectively processes sequential data, adapting well to the complexity of smart contract source codes. Its end-to-end learning approach allows for automatic learning of input-output relationships, reducing manual feature engineering efforts. Furthermore, RNN's scalability makes it suitable for various types of smart contract vulnerability detection tasks.

In addition, some methods warrant analysis. GPTScan is a tool that combines generative pre-trained transformers (GPT) and static analysis [99]. It improves the accuracy and efficiency of detecting logical vulnerabilities by decomposing them into code-level scenarios and properties, using GPT to match candidate vulnerable functions, and further validating them through static confirmation. Its novelty lies in integrating the capability of large language models with program analysis to identify and verify logical vulnerabilities in smart contracts, which is uncommon in previous tools. Besides, Wu's team introduced a vulnerability detection method based on a Hybrid Attention Mechanism Model [100]. The model combines a single-head attention encoder with a multi-head attention encoder, focusing on key vulnerability points by extracting code fragments from the source code, and then employs deep learning techniques for feature learning and vulnerability classification.

2.4.5 Discussion

Fuzzing has a significant effect on smart contract security vulnerability mining. It uses contract runtime information to detect accessible vulnerabilities, thereby improving detection accuracy. However, fuzzing faces challenges in automated testing for contracts without source code or interface information. In addition, the dynamic execution process of fuzzing is relatively complex, and the randomness of test cases limits the ideal test path coverage. Fuzzing, which monitors contracts for abnormal behavior during execution, is effective at discovering vulnerabilities. However, it has limited insight into the specific semantics of the code that underlie

these vulnerabilities. This makes it difficult to locate the exact code location where the vulnerability exists. For example, sFuzz [11], while integrating fuzzing with symbolic execution, may still struggle with complex smart contracts due to the high dimensionality of the search space. ContractFuzzer [74] may be less effective against contracts with intricate logic or requiring complex state changes. ILF [75], which combines neural networks with fuzzing, can improve coverage but may not always provide the most efficient path to finding vulnerabilities, especially in contracts with large and complex state spaces. IR-Fuzz [76], with its data-driven approach, can miss critical state transitions if the data used for training does not adequately represent the contract's behavior.

Symbolic execution identifies contract vulnerabilities by simulating program executions with symbolic variables, and analyzing all possible paths. It enhances path exploration accuracy, reduces computational load by avoiding full EVM simulation, and improves detection by integrating with abstract interpretation and model checking. However, symbolic execution also faces some challenges and limitations. When the program complexity increases, the path will increase exponentially, that is, the path explosion problem, which causes the efficiency of symbolic execution to drop sharply; when the path is too complex, there will also be problems such as timeout or no solution in constraint solving, which reduces the accuracy of symbolic execution; when the program involves uncertainty or randomness, symbolic execution is also difficult to handle. For example, Oyente [10] can face challenges with path explosion, especially in contracts with many conditional branches. Maian's concrete verifier may not always be able to confirm the authenticity of vulnerabilities due to the complexity of the contract's logic [79]. Mythril [71], while powerful, can be limited by the accuracy of the symbolic execution engine and the performance of the underlying solver. Park's parallel-fork symbolic execution can be resource-intensive and may not scale well with the increasing complexity of smart contracts [80].

Formal verification is a method based on mathematical logic and proof. It can verify whether a contract conforms to the expected design and function by defining and reasoning about the properties and specifications of smart contracts. Formal verification can discover various types of vulnerabilities, thereby improving the credibility and reliability of smart contracts. However, formal verification techniques also face some challenges and limitations: it requires manual writing and verification of specifications and properties, which increases workload and error rate, and depends on the personnel's professional level; it is difficult to fully simulate EVM's execution environment and uncertainty or randomness factors, which affects verification accuracy and completeness; it has few open source materials and also lacks relevant supporting tools. For example, Securify's reliance on Datalog language for expressing security properties can be limited by the expressiveness of the language and the complexity of the contract's logic [9]. KEVM [84], while providing executable semantics, may not cover all possible edge cases due to the complexity of the EVM. ZEUS [86], with its static analysis capabilities, can be limited by the accuracy of the abstract interpretation and the complexity of the contract's state space. FSM-based verification [88], while innovative, may struggle with the scalability of verifying large and complex smart contracts due to the combinatorial explosion of states.

In conclusion, traditional contract code automatic detection methods have three main problems:

- First, low degree of automation. The current detection methods rely on manual processing or verification, which increases workload and error rate, and is limited by personnel's technical level and experience.
- Second, poor generalization. Many types of detection tools are currently available, but they cover different types of vulnerabilities and are unstable in accuracy.

- Third, large time overhead. Due to the complexity and diversity of smart contracts, detection tools need to spend a long time analyzing contract code. They may face computational difficulties such as path explosion and constraint solving.

Deep learning enables smart contract vulnerability detection by automatically extracting high-level data features, eliminating the necessity for explicitly defined detection rules, learning features from training data, and diminishing the burden of manual feature engineering. GNNs are specialized in processing graph-structured data, learning node and graph representations from nodes and their neighbors to capture complex relationships, making them suitable for representing the structure and semantics of smart contracts. However, CGE may not be as effective in contracts with non-standard patterns or those that deviate from the expert patterns it was trained on [18]. While Bi-GGNN can be computationally intensive and may require significant training data to achieve high accuracy [18].

CNNs are primarily for grid-structured data, learning high-level abstract representations via multi-layer structures to efficiently process smart contracts. However, CodeNet's reliance on image-based representations may not capture all the nuances of the contract's behavior, especially in contracts with complex data structures [95].

RNNs are apt for processing sequential data, and capturing dependencies in input sequences, particularly suited to the complexity of smart contract source code. BLSTM-ATT [98], while adept at handling sequential data, may struggle with contracts that have long-range dependencies or require an understanding of complex control flows.

In addressing the typical vulnerability case in smart contracts, such as the reentrancy exploited in the DAO attack, the four primary detection methods—fuzzing, symbolic execution, formal verification, and deep learning—demonstrate significant differences in detection efficiency and accuracy:

- The reentrancy takes advantage of improper state management during external calls, and fuzzing may require specific input sequences to trigger such state changes, which can be challenging to generate randomly.
- Symbolic execution detects reentrancy by simulating all possible execution paths of a contract. However, in practice, due to the potential for deep recursion involved in reentrancy, symbolic execution may encounter the path explosion problem, leading to a slow analysis process, especially when the contract logic is complex.
- Formal verification offers a high-accuracy detection method based on mathematical proofs, ensuring that contracts satisfy predetermined security properties across all possible execution paths, theoretically preventing reentrancy entirely. Nonetheless, this method requires precise contract models and expertise, and manually writing and verifying these models can be time-consuming.
- Deep learning identifies reentrancy vulnerabilities by automatically learning contract patterns. It can handle large datasets and shows high accuracy when the training dataset sufficiently covers instances of reentrancy. However, suppose the training dataset lacks sufficient and high-quality samples of reentrancy vulnerabilities. In that case, the model may fail to learn the key features for identifying such vulnerabilities, thereby affecting detection effectiveness.

Upon conducting an investigation of both traditional detection methodologies and deep learning strategies, we have identified a substantial scope for research within the realm of smart con-

tract vulnerability detection. This scope is particularly evident in the realms of code feature extraction, the application of few-shot learning techniques, and the utilization of GNNs. These domains hold considerable promise for augmenting the precision and efficiency of vulnerability detection processes in smart contracts.

2.5 Preliminaries

2.5.1 Representation of Smart Contract

In the application of machine learning to code vulnerability detection, a crucial step involves transforming the code into a vector format to serve as input for the model. When applying machine learning to the detection of vulnerabilities in smart contracts, it is essential to first convert the smart contract code into an appropriate vector representation.

In smart contract vulnerability detection, code representation plays a crucial role. These techniques can be categorized based on the granularity of feature extraction and the abstraction level [101]. Granularity-wise, code representation is divided into token-level, statement-level, and function-level. Token-level is the finest granularity, capturing individual elements of the code but may miss the relationships between them. Statement-level, involving methods like AST [102] and Data Flow Graphs, offers a balance between context and detail, capturing the basic components of code fragments. Function-level representation abstracts higher semantic information but is complex and often involves breaking functions down into smaller segments for analysis.

Regarding abstraction levels, code representation ranges from textual to structural, semantic, and functional levels. Textual-level representation treats code as a sequence of tokens, which can be mapped to vectors using techniques like word2vec [103]. However, this level may lack the structural features of the code. Structural-level representation, such as ASTs, focuses on the syntactic structure, revealing the relationships between statements and expressions. Semantic-level representation goes beyond static structure, capturing execution paths and data flow within the code using techniques like control flow and data flow graphs. Functional-level representation aims to encapsulate the functionality of code segments, preserving syntax and semantics to the fullest.

Each level of code representation has its own advantages and limitations. Token-level representation is straightforward but may miss complex code relationships. Statement-level and function-level representations provide a more detailed understanding of code but require careful analysis to avoid losing essential information. Semantic-level representation, especially with recent advances in graph embedding techniques, is becoming a promising research trend for its ability to capture the nuanced behavior of code. Finally, functional-level representation offers the most comprehensive code view by focusing on the implemented functionalities, regardless of syntactic variations.

In the following sections, we will provide an overview of the representation techniques that will be used in subsequent chapters.

Code2vec

Code2vec [23] is a neural network model for learning distributed representations of code snippets. The purpose of the model is to transform code snippets into fixed-length, continuous-

distributed vectors, which are used to predict the semantic attributes of the code, such as method names. The core idea of the model is to decompose the code snippet into a set of paths in its corresponding AST, and then use an attention mechanism to aggregate the vector representations of these paths, thus generating a code vector.

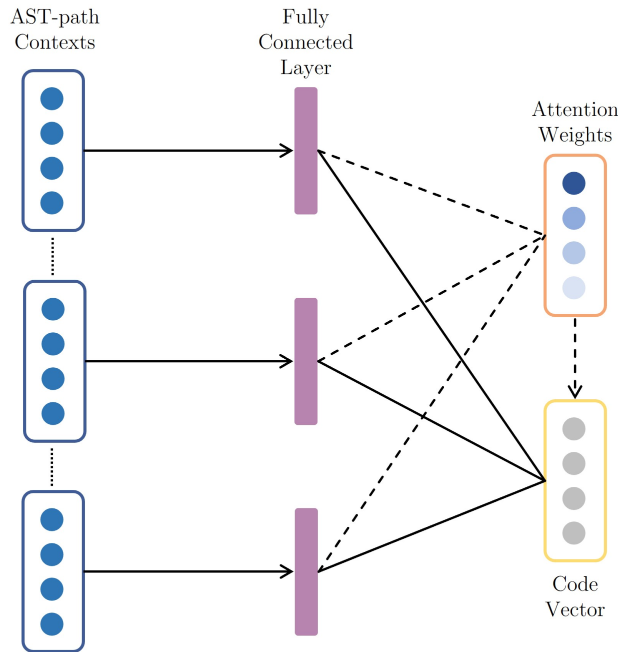


Figure 2.1: Overview of Path-Attention Network.

The research method of code2vec is as follows:

1) Extract paths from the AST of the code snippet. These paths are composed of two terminal nodes and one or more non-terminal nodes, representing the syntactic structure of the code snippet.

2) Represent each path as a path context, which is a triplet containing the values of the path's start and end terminal nodes. Then, use embedding matrices to generate a vector representation for each component of the path context, which is the values of the path's start and end terminal nodes. These vector representations are then concatenated into a vector that represents the entire path context.

3) As shown in Figure 2.1, in order to aggregate these path vectors to generate a code vector, the model adopts a neural network model based on the attention mechanism. The model includes a fully connected layer, which is used to learn how to combine the individual components of the path context vector into a representation. The attention mechanism learns to assign a weight to each path context vector, so that it can be weighted when generating the code vector. This weight is obtained by calculating the dot product between the path context vector and the global attention vector. Finally, the code vector is obtained by adding all the weighted path context vectors.

One-Hot Encode

One-hot encode, a simple and direct method for representing code, assigns each element in a vector to a unique term from a vocabulary, ensuring that only one element is active (set to 1). In contrast, others remain inactive (set to 0). This textual-level approach guarantees orthogonality

between different encodings, eliminating any mutual interference. Initially, one-hot encoding was applied as a sparse representation technique, where the corpus size determined the vector's dimensionality, with each word corresponding to a unique vector dimension. For example, with a corpus like [$apple, pear, orange, banana$], the one-hot vector for *apple* would be [1,0,0,0], *pear* would be [0,1,0,0], *orange* would be [0,0,1,0] and *banana* would be [0,0,0,1]. This example is shown in Figure 2.2.

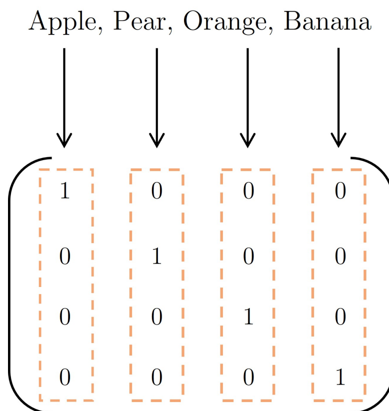


Figure 2.2: Example of One-Hot Encode.

Each category value is represented as a sparse vector in multidimensional space, with the dimensionality equal to the corpus's cardinality. One interesting aspect of one-hot encoding is its representation of categories as independent entities, evident from the zero inner product between any two vectors, signifying equal distance in Euclidean space. As one-hot encoded data is numerical, it easily integrates into machine learning models, allowing for individual parameter learning for each dimension, thus seamlessly incorporating such categorical feature information.

Word2vec

Word2vec [103] is a neural network-based word embedding language model that can map each word in the text to a fixed-dimensional dense vector, quantitatively measure the semantic relationship between words, and predict the vector distribution of each word. This capability is crucial for our method as it enables us to represent the semantic and syntactic structure of Solidity code effectively within the graph representation of smart contracts. Word2vec mainly has two types of architectures: Skip-gram and Continuous Bag-of-Word Model (CBOW). Both types consist of an input layer with one-hot encoding, a hidden layer with linear units without activation function, and an output layer employing softmax regression with dimensionality matching the input.

(1) Skip-gram
It predicts the occurrence probability of the context words corresponding to the input target word by calculating the vector representation of the input target word. Figure 2.3 takes "juice" as an example and introduces the basic structure of the Skip-gram model. In Figure 2.3, the input end is the given word vector v_t, corresponding to the word *juice*, which can predict the occurrence probabilities of the context word vectors v_{t+2}, v_{t+1}, v_{t-1} and v_{t-2}, respectively representing *the*, *apple*, *is*, *delicious*, through a certain calculation of the model.

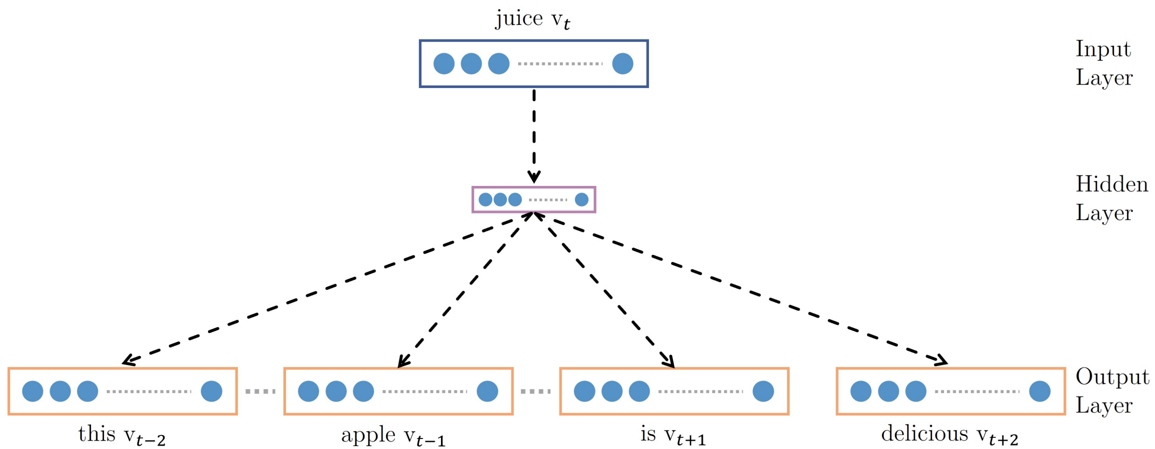


Figure 2.3: Example of Skip-Gram.

(2) Continuous Bag-of-Word
The structure of the CBOW model is exactly the opposite of the Skip-gram model. It predicts the vector representation of the target word based on the context words of the input target word, and then obtains the occurrence probability of the target word. Figure 2.4 takes "juice" as an example and introduces the basic structure of the CBOW model. In Figure 2.4, the input end is the context word vectors v_{t+2}, v_{t+1}, v_{t-1} and v_{t-2} of the given word vector v_t, respectively representing the words *this*, *apple*, *is*, *delicious*, which can predict the target word vector v_t representing *juice* through a certain calculation of the model.

The calculation formula of word2vec is as follows:

$$p(v_i|v_x) = \frac{e^{U_i \cdot V_x}}{\sum_j e^{U_y \cdot V_x}}, \tag{2.1}$$

where vectors V_x and U_y represent the input and output ends, respectively. The model calculates the cosine similarity between the input vector and the output vector, and then normalizes it using the softmax function to derive the conditional probability of word i appearing in the context of word x.

2.5.2 Related Machine Learning Technologies

The impressive successes of machine learning in areas like computer vision, speech recognition, and natural language processing have increasingly drawn researchers' attention to its potential applications in enhancing the security of smart contracts. Aiming at the limitations of traditional detection methods, machine learning methods have the advantage of automatically extracting and abstracting high-level features, avoiding the tedious manual feature extraction process, and having important research value. This thesis aims to explore the smart contract vulnerability detection methods based on machine learning, focusing on few-shot learning and GNN-based detection techniques. By in-depth study of these techniques, we expect to provide more efficient and accurate vulnerability detection methods for smart contract security. This section will introduce GANs and GMNs, which provide the theoretical basis for the methodology of the following chapters.

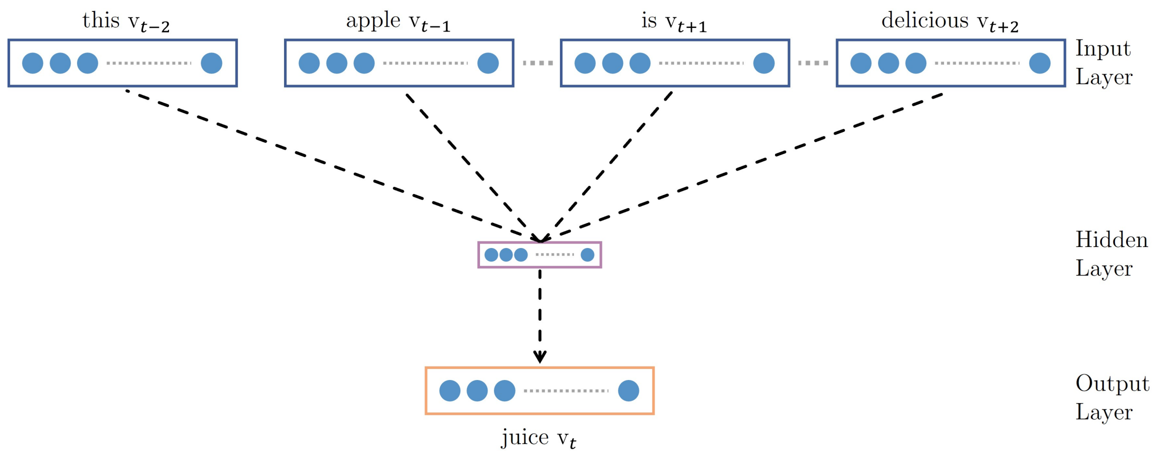


Figure 2.4: Example of Continuous Bag-of-Word.

Generative Adversarial Networks

In the field of machine learning, GANs [22] have become one of the important research areas as an innovative unsupervised learning method. GAN has attracted widespread attention, mainly because it can utilize a large number of unlabeled image datasets, and perform mapping between highly nonlinear data space and latent space. In addition, GAN also shows great potential for semi-supervised or supervised learning in dealing with the problem of insufficient training data.

As shown in Figure 2.5, the core structure of GAN consists of two main parts, namely the generator (G) and the discriminator (D). These two parts are usually implemented by neural networks, although theoretically, any differentiable system that can transform data from one space to another can be used to build these models. The generator's goal is to create new, unseen data instances from random noise, which are similar to the original dataset in terms of visual or statistical characteristics. This means the generator learns to transform noise vectors into vectors with similar statistical characteristics to the real dataset. At the same time, the task of the discriminator is to distinguish whether the samples are from the original dataset or the generator, by analyzing the data samples to estimate the probability of the samples belonging to the original dataset.

The training process of GAN is essentially an adversarial process. In this process, the generator tries to generate more and more realistic data samples to fool the discriminator. In contrast, the discriminator tries to improve its ability to distinguish between real and fake samples. This process can be seen as a minimax game where both parties participate, where the generator tries to increase the error rate of the discriminator, while the discriminator tries to reduce it. The ultimate goal of training GAN is to reach Nash equilibrium, that is, a dynamic balance state where neither party can significantly improve its performance by changing. In this process, the learning rates of the two models need to be properly balanced and adjusted.

- GraphGAN
  In the fields of deep learning and graph representation learning, the proposal of GraphGAN [104] marks important progress in applying GAN to graph structure data. The main goal of GraphGAN is to learn the low-dimensional representation of graph structure data by integrating the generative model and the discriminator into an adversarial framework. In order to overcome the limitations of the traditional softmax function in graph representation learning, GraphGAN also proposes a novel graph softmax function,

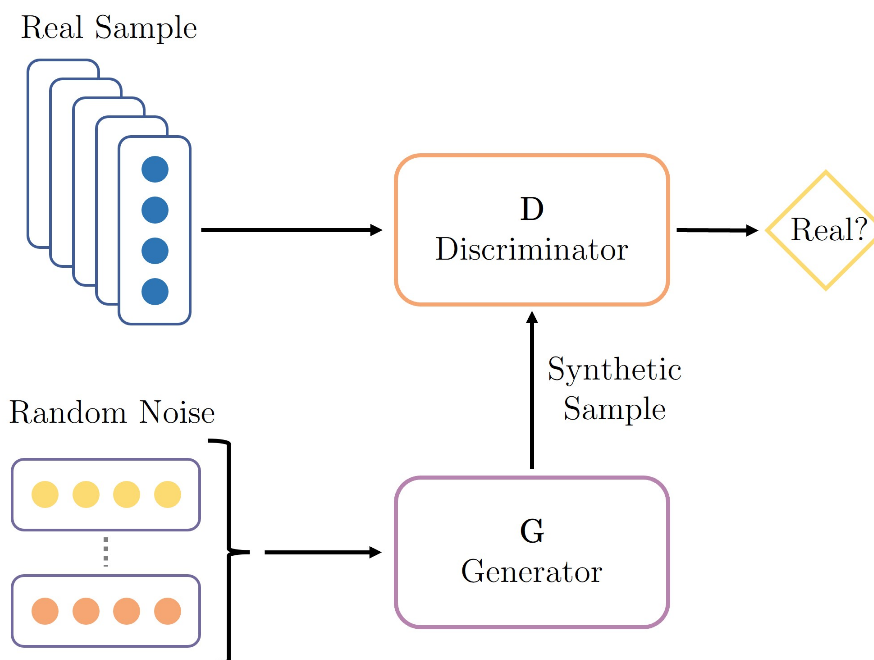


Figure 2.5: Structure of Generative Adversarial Networks.

aiming to improve the performance of graph representation learning.

The core methodology of GraphGAN is to combine the generative model and the discriminator in a unified adversarial framework, making them compete with each other to improve performance. In this framework, the generator strives to adapt to the node distribution of the real graph and generates synthetic nodes to deceive the discriminator. Meanwhile, the discriminator tries to accurately distinguish the generated nodes and the real nodes in the graph, and make judgments on the edge status between the nodes. In addition, the graph softmax function proposed by GraphGAN has the characteristics of normalization, graph structure awareness, and high computational efficiency, which effectively overcomes the shortcomings of the traditional softmax function. GraphGAN also adopts a random walk strategy to generate samples, aiming to reduce the computational complexity and improve the sample quality.

GraphGAN shows significant performance improvement in various graph representation learning tasks. These tasks include data augmentation, graph reconstruction, link prediction, node classification, recommendation systems, and data visualization. The promising applicability and proven efficacy of GraphGAN in graph representation learning lay the technical groundwork for our subsequent exploration into a graph-based few-shot learning detection model.

Graph Matching Networks

Graph Matching Networks [19] is a GNN [90] model for learning graph structure data similarity. Its research aims to solve graph structure data's similarity learning problem, especially in computer security, such as binary function similarity search and other tasks. GMN can help identify binary files with similar functions by learning the similarity of graph structure data, thus discovering potential security vulnerabilities.

Figure 2.6 shows the overview of GMNs. Unlike GNNs, the GMN not only learns the rep-

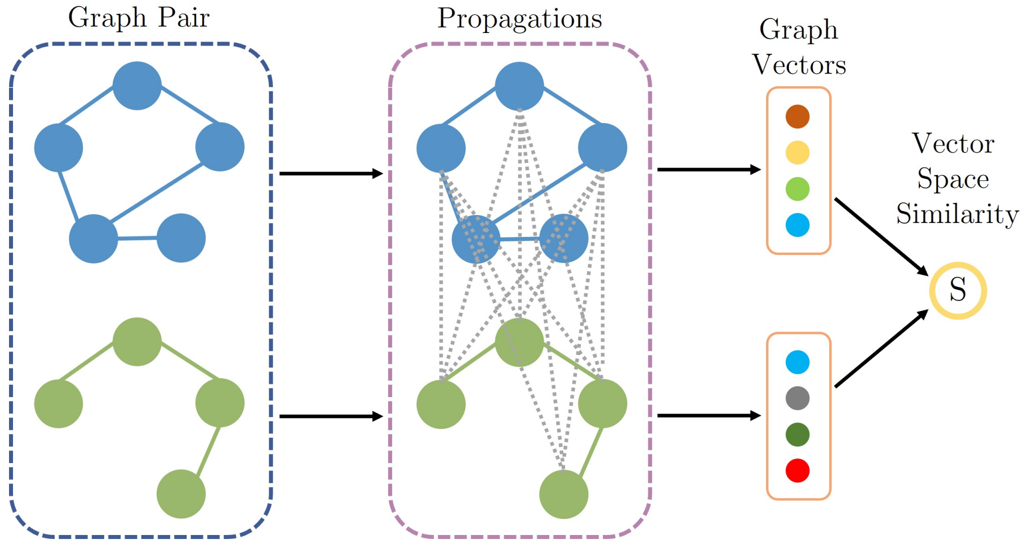


Figure 2.6: Overview of Graph Matching Networks.

resentation of each graph, but also jointly learns the representation of two graphs through a cross-graph attention mechanism. This mechanism enables GMN to capture the similarity information between graphs more effectively. The computation process of GMN includes the following steps:

1) Encoding layer: Maps the features of nodes and edges to initial node and edge representations.

2) Propagation layer: Aggregates local structure information to node representations via multi-layer propagation and aggregation operations, that is, updates the node and edge information in the graph by using a cross-graph attention mechanism.

3) Aggregation layer: Aggregates the node vectors and edge vectors to obtain a whole graph vector containing the complete semantic and syntactic information of the graph.

4) Matching function: Calculates the similarity score between two graphs according to the attention weights.

5) Loss function: In order to train the GMN model, a suitable loss function needs to be designed. Common loss functions include pairwise loss and triplet loss. Pairwise loss is used to optimize the distance between similar graph pairs. In contrast, triplet loss is used to optimize the distance between similar graph pairs smaller than the distance between dissimilar graph pairs.

GMN has achieved significant performance improvement on multiple graph similarity learning tasks. Compared with graph kernel-based methods (Weisfeiler-Lehman kernel [105]), GMN can better capture the graph structure information and semantic information, thus achieving more accurate similarity learning. GMN provides a new solution for graph structure data similarity learning. Using a cross-graph attention mechanism, GMN can capture the similarity information between graphs more effectively, thus achieving more accurate similarity learning. GMN provides us with new ideas for designing smart contract vulnerability detection.

2.6 Conclusion

This chapter mainly introduces the related theories and techniques involved in this thesis. First, the technical principle and security status of Ethereum smart contracts are introduced, and ten types of vulnerabilities in three levels are summarized, and their causes are analyzed. Second, the current mainstream smart contract vulnerability detection methods are classified and analyzed, and the shortcomings of traditional methods are pointed out. While studying related works, we found the potential of machine learning in the smart contract vulnerability detection field. Then, we introduce and analyze the code representation, few-shot learning, and GNN techniques in machine learning, which lay the theoretical foundation for the research in the following chapters.

Chapter 3

Detection Method Based on Code Representation and Generative Adversarial Networks

3.1 Introduction

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 [39]. However, smart contracts face increasingly serious security issues due to their unique operating environment and programming characteristics. The BEC Token Attack and the Proof of Weak Hands (PoWH) incident [106] underscore the critical importance of addressing vulnerabilities in smart contracts. In April 2018, a hacker exploited an integer overflow in the BEC token's smart contract, resulting in the creation of an excessive number of tokens and a subsequent market crash. Similarly, the PoWH contract suffered the same vulnerability, leading to the loss of significant Ether. These incidents underscore the critical need for our detection method to concentrate on integer overflow. We focus on Ethereum-based smart contracts and propose a high-precision and versatile detection method to address the integer overflow vulnerability.

The construction of a vulnerability detection model still faces some challenges, especially in collecting smart contract source code for experiments. Related research shows that the proportion of publicly available smart contract source code is very low, only about 1% [20]. At the same time, given the limitations of the Ethereum network and nodes, obtaining a large number of applicable source codes requires a lot of time and resources. It is essential to screen the dataset manually to ensure its quality and security. In addition, acquiring real vulnerability data may involve privacy and legality issues. In order to build an effective code representation and machine learning-based vulnerability detection model, we need to pay special attention to the quality of the training samples and the scale of the data set. High-quality samples are essential for enhancing the accuracy and generalization ability of the model. The lack of data may limit the model's performance in detecting vulnerabilities. In view of these challenges, we are inspired by data augmentation techniques in traditional machine learning to explore and develop a few-shot learning method suitable for smart contract vulnerability detection.

We innovatively apply Generative Adversarial Network (GAN) technology [22] to smart con-

tract vulnerability detection. GAN consists of two parts: a generator and a discriminator. The generator is responsible for generating data, while the discriminator tries to distinguish between the generated and real data. In our application, the generator of GAN is used to generate synthetic contracts that are infinitely close to real smart contracts, which is particularly important for expanding the data set. The discriminator is trained to distinguish between real contracts and synthetic contracts accurately. This method can solve the possible data shortage problem. Specifically, we use a code embedding algorithm to convert the smart contracts source code into spatial vectors, thereby retaining as much syntactic and semantic information as possible. Based on this, we use GAN to train small sample vector data sets to generate a large number of synthetic data sets for similarity judgment, thereby achieving the purpose of small sample detection. Our proposed model combines the GAN discriminator feedback and vector similarity analysis to identify smart contracts containing integer overflow vulnerabilities. The model combines the adversarial training mechanism of GAN and the key feature extraction of smart contracts, thereby achieving higher accuracy and efficiency in the security analysis of smart contracts.

3.2 Methodology Framework

The GAN-based smart contract integer overflow vulnerability detection system mainly consists of three stages: source code preprocessing, model training, and similarity detection. As shown in Figure 3.1, first, the source code is preprocessed according to the integer overflow vulnerability features, removing useless and interfering fragments, and generating corresponding Abstract Syntax Trees (AST) [102]. Then, code2vec [23] is used to convert ASTs into contract vectors. This preprocessing process is for all smart contracts, ensuring data consistency. The preprocessed training set will be used for GAN model training, where the generator and discriminator compete with each other, the generator tries to generate more realistic synthetic contracts, and the discriminator tries to distinguish real and synthetic contracts more accurately. Finally, the generator can generate high-quality synthetic samples with the target data distribution characteristics. The trained generator will be used to expand the sample size of the test set.

To detect whether a smart contract contains an integer overflow vulnerability, the contract source code needs to be preprocessed and converted into a corresponding vector, and then the contract vector is input into the trained discriminator to obtain the corresponding security label. If the label is positive, the contract vector is compared with the expanded test set for vector similarity calculation. The detection system determines whether the contract contains vulnerability based on the preset similarity threshold coefficient. The contract contains the vulnerability if the similarity coefficient exceeds the threshold; otherwise, the contract does not contain the vulnerability.

The workflow of the detection system can be summarized as follows:

1) Source code preprocessing: Preprocess and generate contract vectors according to the vulnerability features.

2) Model training: Use GAN to train the generator and discriminator. The generator generates high-quality synthetic contracts, and the discriminator distinguishes real and synthetic contracts.

3) Vulnerability detection: Input the target contract vector into the discriminator. If the label is positive, perform vector similarity calculation, and determine whether it contains

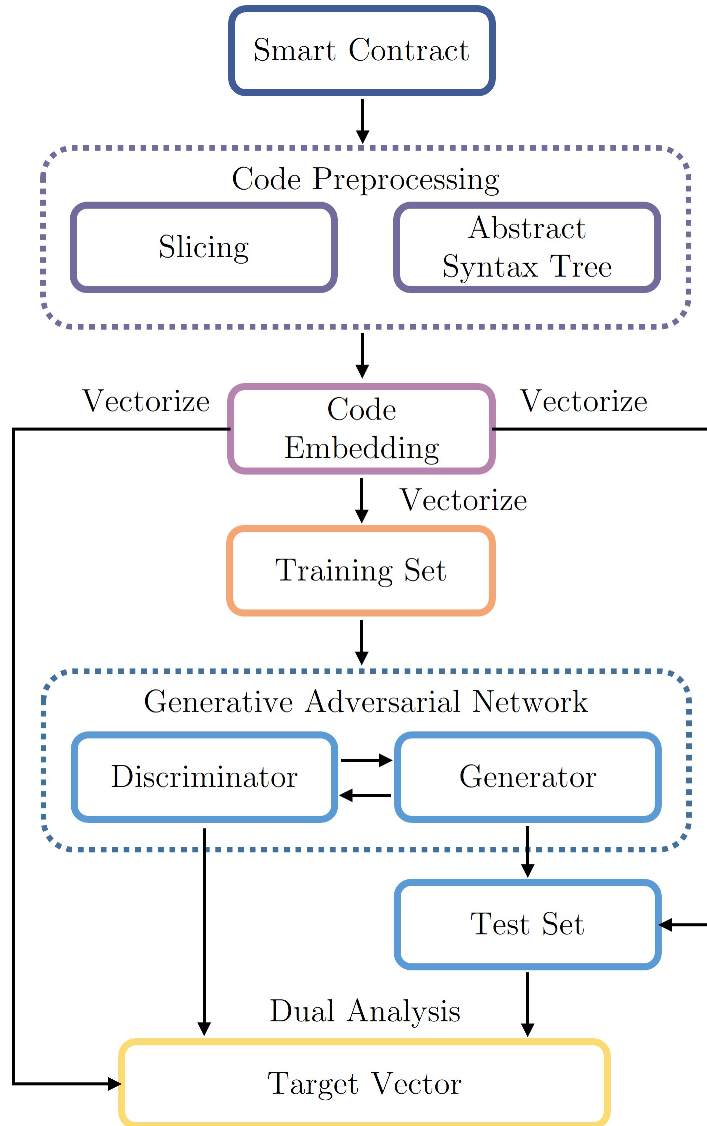


Figure 3.1: Generative Adversarial Networks-based Detection Method Framework.

the vulnerability according to the similarity threshold coefficient.

3.3 GAN-based Data Augmentation

Figure 3.2 illustrates the process of generating vulnerable contract data, which consists of three stages: Code Preprocessing, Code Embedding, and Code Generation.

3.3.1 Code Preprocessing

Smart contract source code often contains sensitive information such as transaction details and user data. It may violate data protection laws if used for model training without proper processing. As discussed in Section 1.2, we collect secure open-source data for model training and testing.

Furthermore, the smart contract programming language Solidity allows developers to customize identifiers such as function names, variable names, etc. Different developers may have different

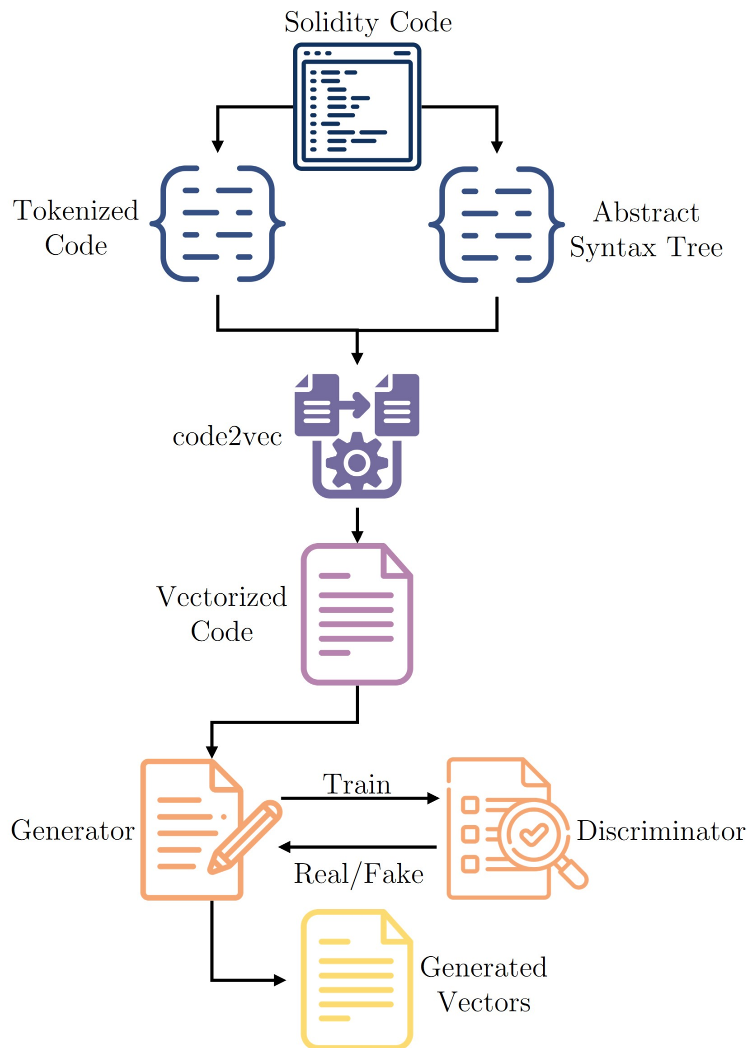


Figure 3.2: Process of Data Generation.

coding styles, such as naming conventions, programming habits, etc. This may cause semantically equivalent code fragments to be converted into different vectors when the source code is vectorized, affecting the training effect of the GAN model and the accuracy of similarity judgment. Therefore, we need to preprocess the source code. Preprocessing includes the following rules:

- Preserve the key features of integer overflow vulnerability.
- Preserve the code semantic and structural features.
- Follow the input specification of code embedding.

Therefore, we will standardize from the following two aspects.

Code Feature Specification

It is essential to establish the features of integer overflow. The main cause of integer overflow is the design of integer types in Solidity, a smart contract programming language. In EVM, integers are unsigned and fixed-size data types, whose range is determined by their bit width. Solidity supports integer types from uint8 to uint256, where uint8 represents an 8-bit unsigned

integer and uint256 represents a 256-bit unsigned integer. As shown in Figure 3.3, when a uint8 variable stores the number 255 and performs an addition operation on it, the result will exceed the maximum value of uint8, causing an overflow and becoming 0. Table 3.1 shows the key features of integer overflow, which must be highlighted in the source code.

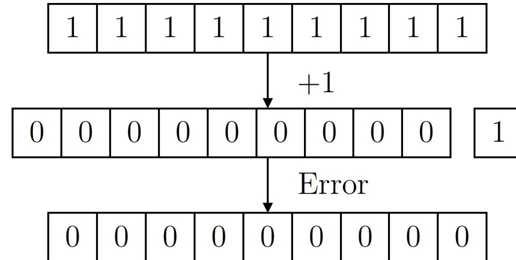


Figure 3.3: Principle of Integer Overflow.

Table 3.1: Example of Integer Overflow Features.

Vulnerability	**Key Features**					
Integer Overflow	*add*	*mul*	*sub*	*unit*	$\cdots$	*int*

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 [107]. To capture the syntax of the code, we use the solidity-parser-antlr [102] to generate the AST of the code, and assign a unique identifier to each node of the AST to differentiate between different node types. The following is a simple smart contract *Sol* and its corresponding AST, which is shown in Figure 3.4.

Listing 3.1: An Example of Smart Contract *Sol*.

```
1 contract Sol {
2     uint256 public x = 2023;
3     function test(uint8 y) public {
4         x += y;
5     }
6 }
```

Embedding Input Specification

The AST generated by solidity-parser-antlr needs to be further processed to meet the embedding requirements of code2vec. More specifically, the traversable AST that meets the embedding requirements needs to follow the following definition:

- We define the AST of a smart contract as $\langle N, T, X, s, \delta, \phi \rangle$, where N is the set of non-terminal nodes, T is the set of terminal nodes, X is the set of values, $s \in N$ is the root node, $\delta : N \to (N \cup T)^*$ is a function that maps non-terminal nodes to their list of child

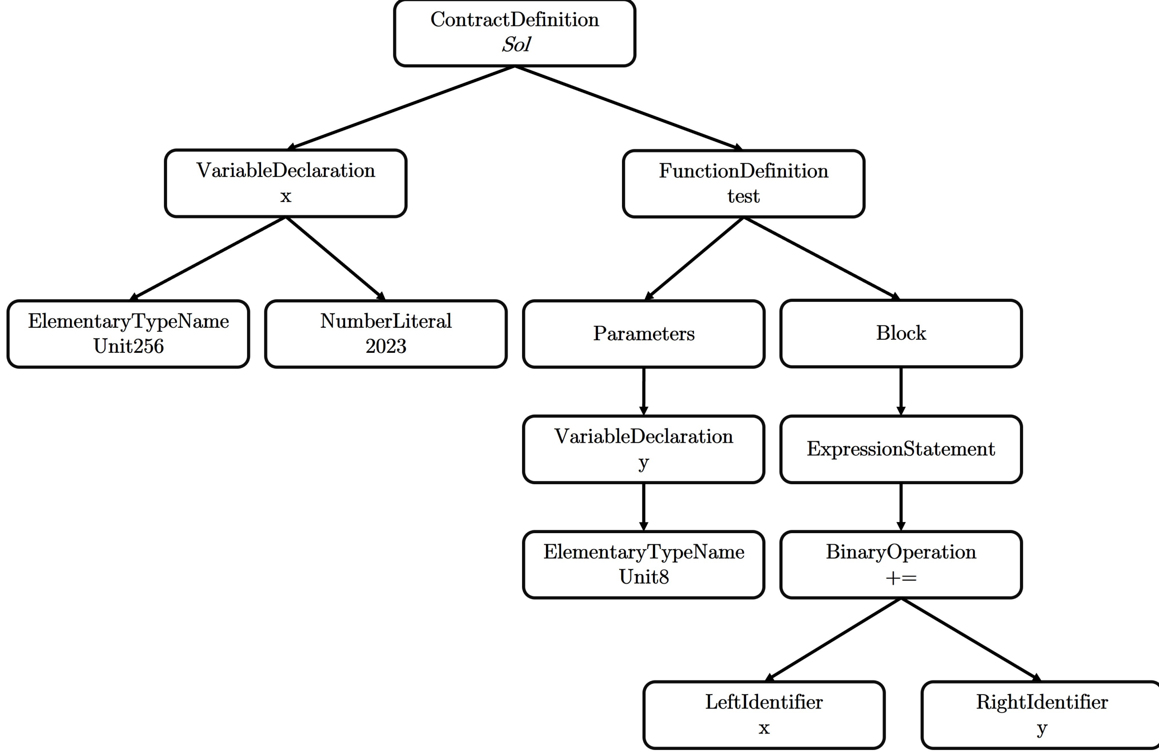


Figure 3.4: Abstract Syntax Tree of Smart Contract *Sol*.

nodes, $\phi : T \rightarrow X$ is a function that maps terminal nodes to their corresponding values. Except for the root node, each node appears only once in all child node lists.

- A path in an AST of length λ can be defined as a sequence $n_1 d_1 ... n_\lambda d_\lambda n_{\lambda+1}$, where $n_1, n_{\lambda+1} \in T$ are terminal nodes and $n_i \in N$ for $i \in [2..\lambda]$ are non-terminal nodes. The movement direction in the AST is represented by $d_i \in \{\uparrow, \downarrow\}$ for $i \in [1..\lambda]$. When $d_i = \uparrow$, it implies $n_i \in \delta(n_{i+1})$, and for $d_i = \downarrow$, it denotes $n_{i+1} \in \delta(n_1)$. For any given path p, the starting node n_1 is denoted by $s(p)$, and the terminal node $n_{\lambda+1}$ is represented by $e(p)$.
- For a given AST-path p, the context of the path is defined as a triplet $< x_s, p, x_t >$, where

$$x_s = \phi(s(p)), \tag{3.1}$$

represents the value of the starting node and

$$x_t = \phi(e(p)), \tag{3.2}$$

corresponds to the value of the terminal node.

We demonstrate with the smart contract *Sol*. Based on Figure 3.4, we marked out two AST-paths $P1$ and $P2$ with blue and red arrows, respectively. As shown in Figure 3.5, these paths consist of $\uparrow$ and $\downarrow$ arrows and connected nodes, indicating the traversal direction between adjacent nodes in the AST. Among them:

$$P1 = (VD \uparrow CD \downarrow FD \downarrow Bl \downarrow ES \downarrow BO \downarrow); \tag{3.3}$$

$$P2 = (VD \uparrow Pa \uparrow FD \downarrow Bl \downarrow ES \downarrow BO \downarrow). \tag{3.4}$$

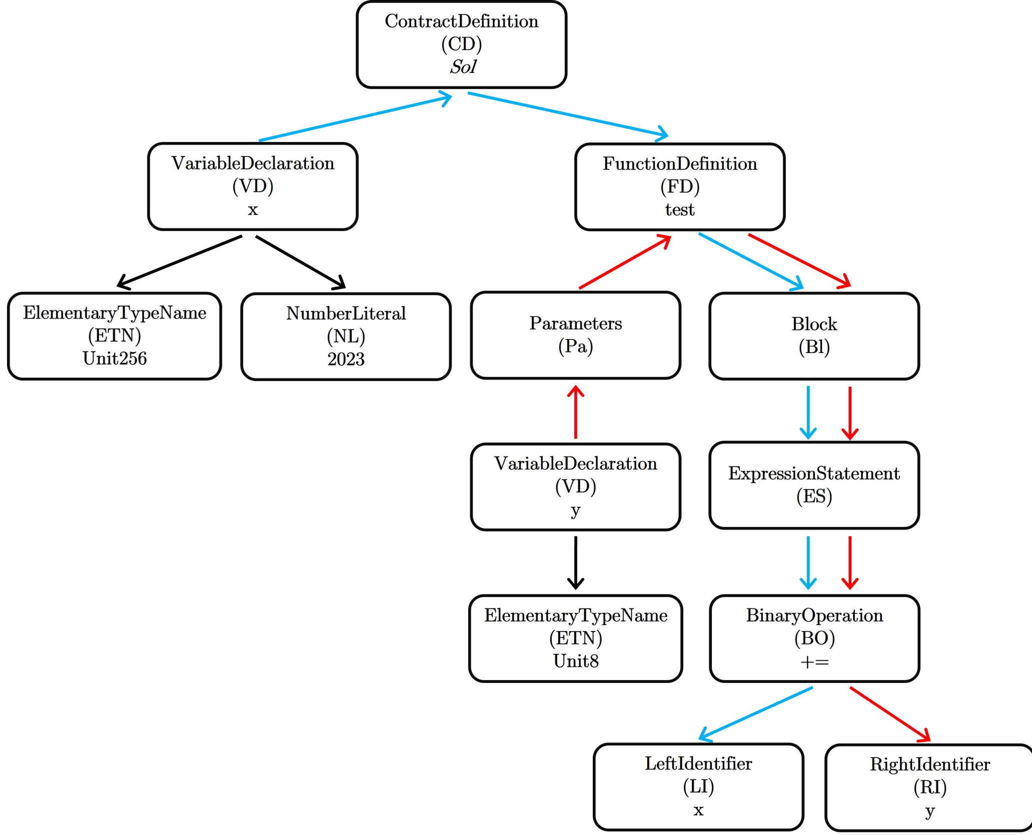


Figure 3.5: Example of Abstract Syntax Tree Preprocessing.

For $P1$ and $P2$, their corresponding path-contexts between the starting node and the terminal node are as follows:

$$P - Context - 1 =< VD(x), P1, LI(x) >; \tag{3.5}$$

$$P - Context - 2 =< VD(y), P2, RI(y) > . \tag{3.6}$$

3.3.2 Code Embedding

After completing the semantic analysis, we train the code using code2vec to obtain corresponding vectors. To be specific, 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. As shown in Figure 3.6, 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:

$$c = \sum_{i=1}^{n} a_i p_i; \tag{3.7}$$

$$p_i = H(s_i, t_i, r_i). \tag{3.8}$$

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 ith 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 generate all possible AST-path contexts to represent the semantic information of the code.

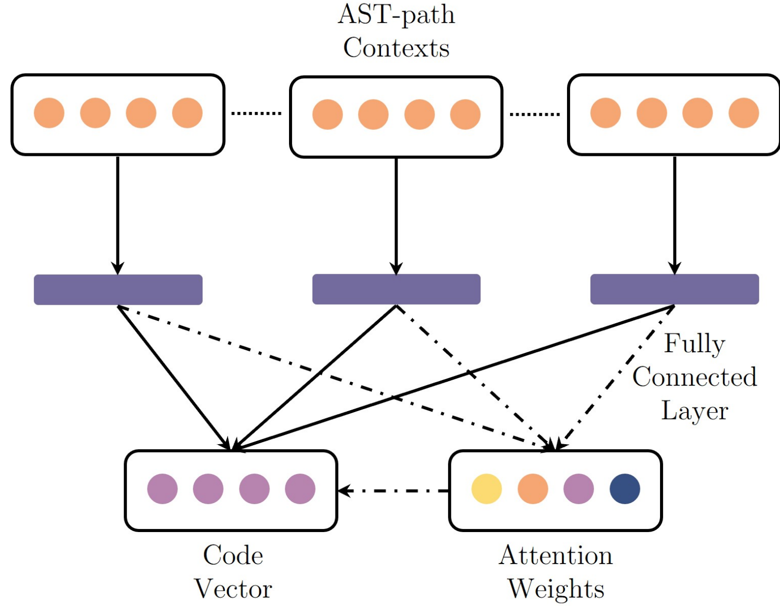


Figure 3.6: Overview of Code Embedding.

3.3.3 Code Generation

As shown in Figure 3.7, 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 [108], [109] 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:

$$L_G = -E_{z \sim p(z)}[\log D(G(z))]; \tag{3.9}$$

$$L_D = -E_{x \sim p_{data}(x)}[\log D(x)] + E_{z \sim p(z)}[\log(1 - D(G(z)))]. \quad (3.10)$$

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 [104]. 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.

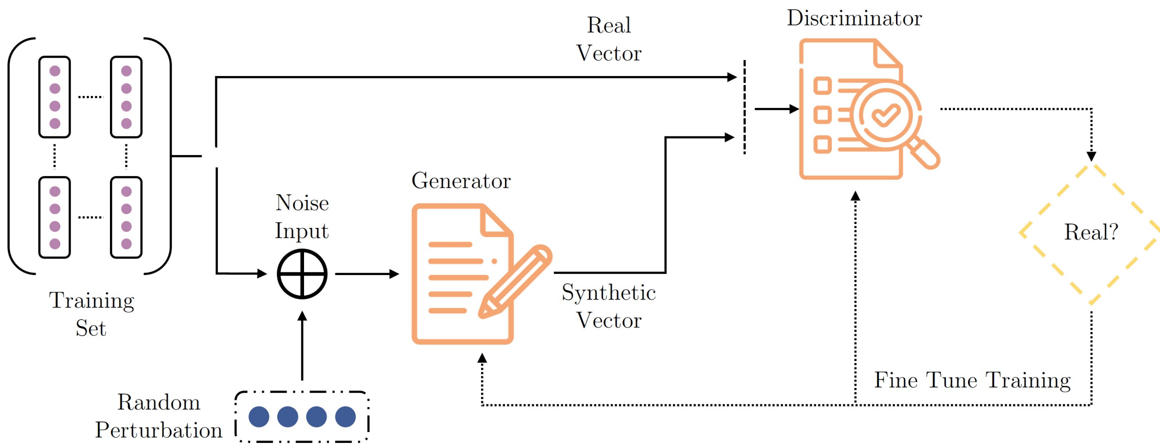


Figure 3.7: Process of Synthetic Code Generation.

3.4 Dual Similarity Detection

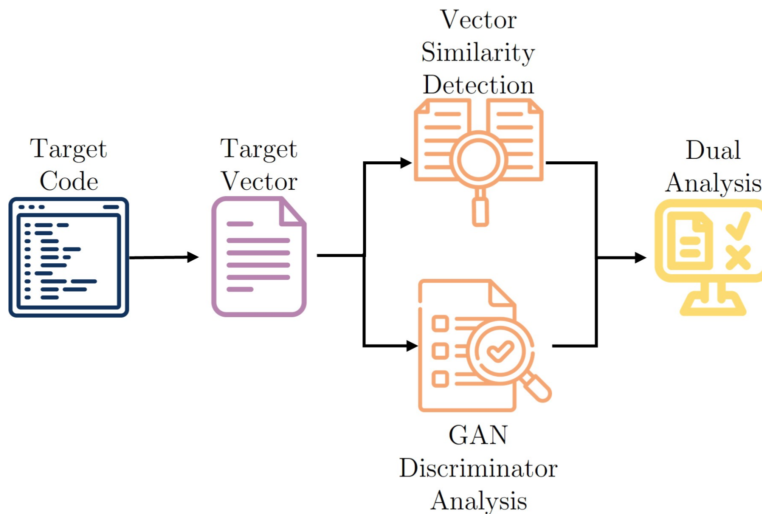


Figure 3.8: Overview of Similarity Detection.

Figure 3.8 illustrates the overview of vulnerability detection. This dual detection method can be divided into two parts: GAN discriminator analysis and vector similarity analysis.

3.4.1 GAN Discriminator Analysis

During the GAN training process in Section 3.3.3, 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. As shown in Algorithm 1, 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.

Algorithm 1 GAN Discriminator Analysis

Input:
Target code, TC;
Preprocess, P;
Embedding, E;
Discriminator, D.

Output:
Target AST, TA;
Target vector, TV;
Vulnerability label, VL.

1: **procedure** DISCRIMINATOR ANALYSIS
2: $TA \leftarrow Preprocess(TC, P)$
3: $TV \leftarrow Embedding(TA, E)$
4: $VL \leftarrow Discriminator(TV, D)$
5: **if** $VL =$ True **then**
6: **Target code contains a vulnerability.**
7: **else** $VL =$ False
8: **Target code does not contain a vulnerability.**
9: **end if**
10: **end procedure**

3.4.2 Vector Similarity Analysis

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 [110]. 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 [17] and correlation coefficient [111] are used to calculate the vector similarity between the target and vulnerable contracts. The specific formula is as follows:

$$\cos(x, y) = \frac{1}{n} \sum_{i=1}^{n} \frac{x \cdot y_i}{\|x\| \|y_i\|}; \tag{3.11}$$

$$r = \frac{1}{n} \sum_{i=1}^{n} \frac{\sum_{j=1}^{m} (x_j - \bar{x})(y_{ij} - \bar{y}_i)}{\sqrt{\sum_{j=1}^{m} (x_j - \bar{x})^2 \sum_{j=1}^{m} (y_{ij} - \bar{y}_i)^2}}, \tag{3.12}$$

where x represents the vector of the target contract, and y denotes the vector collection of the vulnerable contracts. Let y_i be the ith 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 the appropriate weight and threshold in Section 3.5. By combining these two analysis methods, we can significantly improve vulnerability detection accuracy.

3.5 Experimental Result and Analysis

In previous Sections, we introduced how to transform the smart contract source code into vectors that contain structural and semantic features, and use the GAN model to perform data augmentation to expand the vulnerable contract dataset. We also introduced how to combine the GAN discriminator and vector similarity to judge whether the smart contract has integer overflow vulnerabilities. This section is dedicated to validating the efficiency and benefits of the proposed approach in identifying integer overflow vulnerabilities in smart contracts, utilizing experiments. We will first describe the experimental environment and dataset, then determine the optimal threshold coefficient for vector similarity through experiments, and compare and analyze with other related tools.

3.5.1 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 use solidity-parser-antlr [102] to generate ASTs from the smart contract source code, and use code2vec [23] to extract the corresponding feature vectors from the ASTs. To evaluate the effectiveness and advantages of the proposed method in detecting integer overflow vulnerabilities in smart contract source code, we designed the following two experimental steps:

1) Vector similarity parameters selection: In Section 3.4.2, we proposed a vector similarity analysis method based on cosine similarity and correlation coefficient, which uses the weighted average of cosine value and Pearson correlation coefficient as the measure of vector similarity. To determine the optimal weights and thresholds, we experimented with these two parameters, observed their impact on the accuracy and recall of vulnerability detection, and selected the parameters that are most suitable for smart contract source code vulnerability detection.

2) Vulnerability detection effectiveness evaluation: We collected smart contracts with security labels from the open-source platform, formed our base dataset, and divided it into training and testing sets. Then, we compared the proposed method with two existing detection tools analyzed in Section 2.4, to demonstrate the effectiveness and advantages of our method.

3.5.2 Dataset and Evaluation Metric

This section describes our data collection method, evaluation metric selection and experimental comparison setting, aiming to validate the performance of the proposed vulnerability detection method.

Dataset

In order to assess the efficacy and benefits of our proposed method for detecting integer overflow vulnerabilities in smart contract source code, we collected a dataset of 200 smart contracts sourced from the Etherscan platform. Etherscan is an Ethereum block explorer and analytics platform [112]. These smart contracts all contain solidity source code, contract address and security attribute label, for us to conduct further analysis and evaluation. We divided these smart contracts into two datasets, as shown in Table 3.2: the training set contains 50 smart contracts with integer overflow vulnerabilities, which are used to train the GAN model and vector similarity analysis. The testing set contains 150 smart contracts, of which 70 have integer overflow vulnerabilities, while the remaining 80 are safe contracts. The testing set is used to evaluate the detection performance of our method and other tools.

Table 3.2: Summary of Original Dataset

Type of Dataset	**Number of Smart Contracts**	
	Safe	**Integer Overflow**
Training Set	0	50
Test Set	80	70

In addition, we also use the trained GAN model to generate 1,950 synthetic contracts. Together with the 50 real contracts in the training set, these form the analysis dataset, which is used for vector similarity detection, as shown in Table 3.3.

Table 3.3: Summary of Augmented Dataset

Type of Dataset	**Number of Smart Contracts**	
	Safe	**Integer Overflow**
Training Set	0	50
Synthetic Set	0	1950
Analysis Set	0	2000

Evaluation Metric

After preparing the datasets, we selected two smart contract vulnerability detection tools, sFuzz [11] and Oyente [10] to conduct the experimental comparison. The selection of these two tools is based on the following principles:

- The source code of the detection tool is publicly available.

- Many vulnerability detection tools have chosen this tool as the testing benchmark when testing their own effectiveness.
- This tool can detect the type of vulnerability that we are concerned with.

We used these two tools to perform vulnerability detection on the test set. Subsequently, we analyzed, evaluated, and compared the detection results to demonstrate the effectiveness and advantages of the vulnerability detection method proposed in this chapter. When testing the performance of each tool, we used Confusion Matrices [113] to represent the detection results, as shown in Table 3.4.

Table 3.4: Confusion Matrices

Security Label	**Detection Result**	
	Safe	**Vulnerability**
Safe	TN	FP
Vulnerability	FN	TP

Based on this, we use Accuracy(ACC) [113] as the evaluation metric for the detection model:

$$ACC = \frac{TP + TN}{TP + TN + FP + FN}. \tag{3.13}$$

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 ACC, the higher the precision of the model.

3.5.3 Vector Similarity Parameter Experiment

The parameters of the composite vector similarity detection method have an impact on the smart contract detection performance, so this section will explore the optimal values of the parameters through experiments. More specifically, we will adjust the weight of cosine similarity and the vector similarity threshold, respectively, and observe their impact on the detection performance to select the best parameter combination for integer overflow vulnerability detection, so that the proposed method can achieve the best performance in detection.

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

$$S = \frac{\cos(x, y) \cdot W + r \cdot (1 - W)}{2}. \tag{3.14}$$

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 weight value to observe the model accuracy change, as shown in Figure

3.9. Through experiments, we set W to 0.74, at which the model achieves the highest accuracy. The experimental results also show that cosine similarity plays a significant role in detecting integer overflow vulnerabilities. We made the following inferences based on the experimental results:

- Code structure and semantic information: cosine similarity can capture the structural and semantic similarity between codes well. Integer overflow vulnerabilities tend to have similar code structures and semantic features. Therefore, by calculating the cosine similarity between codes, these vulnerabilities can be effectively detected.
- Robustness: cosine similarity has strong robustness to noise and outliers. In practical applications, there may be some trivial differences in the code, such as comments, spaces, etc. Cosine similarity can ignore these differences to some extent, thereby improving the robustness of the model.

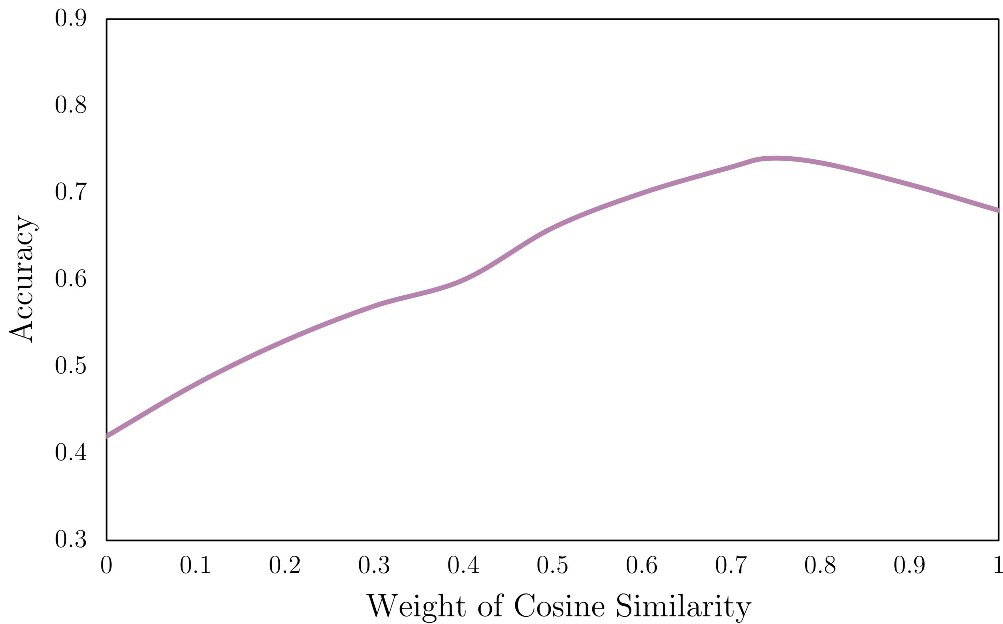


Figure 3.9: Experiments on the Weight of Cosine Similarity.

After determining the value of W, we also need to determine the value of the threshold T through experiments. The choice of the threshold T directly affects the model's sensitivity to vector similarity. When the threshold is high, the model has a high requirement for vector similarity, which improves the accuracy of the model. However, too high a threshold may increase the false positive rate, while too low a threshold may increase the false negative rate. In addition, the choice of the threshold also affects the complexity of the model, too high or too low a threshold may reduce the performance of the model. The experimental results are shown in Figure 3.10. In order to balance the false positive rate and the false negative rate, we set the threshold of the model to 0.9, which ensures the detection accuracy and also has good generalization ability.

3.5.4 Detection Accuracy Experiment

To evaluate the detection performance of the proposed method and other tools, we use the testing set as the dataset for the comparative experiment, which contains 70 contracts with

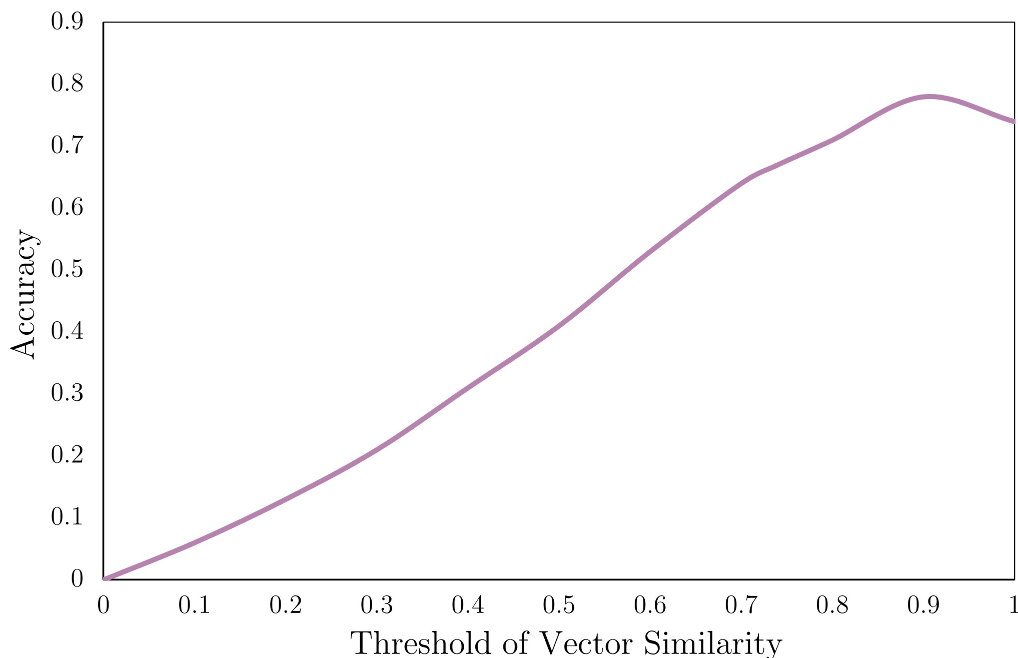


Figure 3.10: Experiments on the Threshold of Vector Similarity.

integer overflow vulnerabilities and 80 safe contracts. We first place the contracts in the testing set into different folders according to their security type; then, we run our model, sFuzz and Oyente, to detect the contracts in each folder; in addition, to verify the detection capability of dual analysis, we also set up two experimental groups: one group only use GAN discriminator analysis (model α), and the other group only use vector similarity detection (model β). Finally, we compare the output results of each detection model with the original contract labels, and obtained the number of TP, FP, TN and FN for each detection model. Based on these numbers, we calculate the accuracy of each detection tool.

The detection results are shown in Figure 3.11. 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 and vulnerable 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.

From the experimental results, it is obvious that our model based on few-shot learning achieved 78% accuracy in detecting integer overflow vulnerabilities, which is close to Oyente, and outperforms the other three models in detection performance. In addition, we also found that sFuzz has some limitations in handling complex semantic features, and its detection effect for integer overflow vulnerabilities is not ideal. In summary, our model achieved the purpose of effective vulnerability detection on a small sample training set, and showed good performance in detecting integer overflow vulnerabilities.

3.5.5 Detection Efficiency Experiment

We conduct experiments to measure the efficiency of a model by calculating the average detection time for each contract. As shown in Figure 3.12, it is obvious that our model has much higher detection efficiency than Oyente and sFuzz. Based on the experimental results, we drew

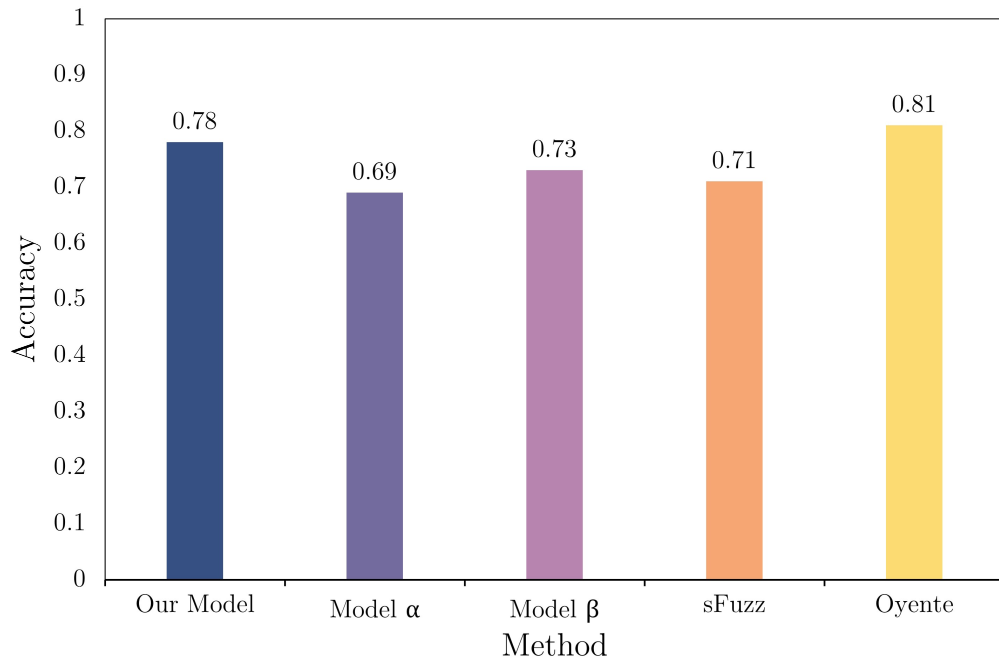


Figure 3.11: Methods Accuracy Experiment.

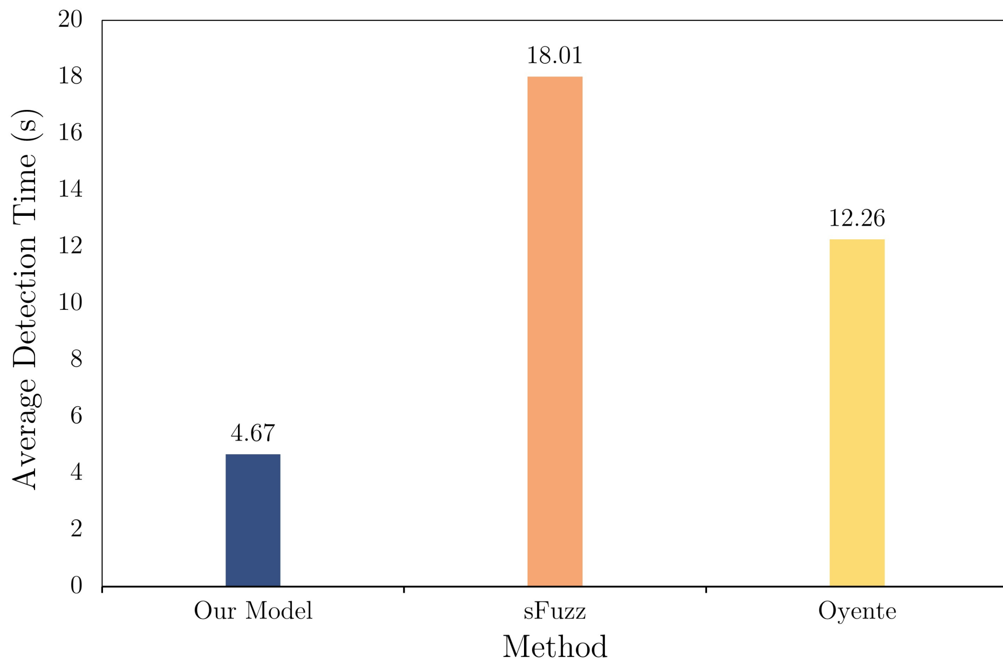


Figure 3.12: Methods Efficiency Experiment.

the following analytical conclusions:

- Oyente uses the symbolic execution technique, which builds and supplements control flow graphs, uses Z3 solver to solve conditional jumps, and determines execution paths based on the solution results to detect vulnerabilities in smart contracts. However, this technique has limitations in handling complex semantic features (such as environmental variables, library functions and dynamic calls), resulting in low detection efficiency.
- sFuzz uses the fuzzing technique, which generates invalid, unexpected or random data as input to detect vulnerabilities in smart contracts. sFuzz requires preprocessing contracts, including building control flow graphs and collecting function dependencies, and then generating input data according to random mutation and coverage feedback strategy. This preprocessing technique is slow in handling complex semantic features, which further affects the detection efficiency.
- Our model uses the code embedding technique, which transforms smart contracts into concise vector representations, and retains the key structural and semantic information. This helps to improve the efficiency and effectiveness of the model in detecting vulnerabilities.

3.6 Conclusion

In this chapter, 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 few-shot 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.

Chapter 4

Vulnerability Detection Method Based on Generative Adversarial Networks and Graph Matching Networks

4.1 Introduction

With Blockchain technology's decentralized and tamper-proof characteristics, smart contracts have been developed rapidly for wide application in critical areas, e.g., the Internet of Things, digital management, healthcare, and finance. However, the security vulnerabilities of smart contracts have led to significant economic losses. Once deployed on the blockchain, smart contracts cannot be modified, making pre-deployment vulnerability detection crucial. We focus on Ethereum-based smart contracts and innovatively propose a detection method based on Generative Adversarial Networks (GAN) [114] and Graph Matching Networks (GMN) [19] to uncover reentrancy and integer overflow vulnerabilities, which greatly impact the security of smart contracts. It is worth mentioning that this is the first method that utilizes these two techniques to achieve smart contract vulnerability detection.

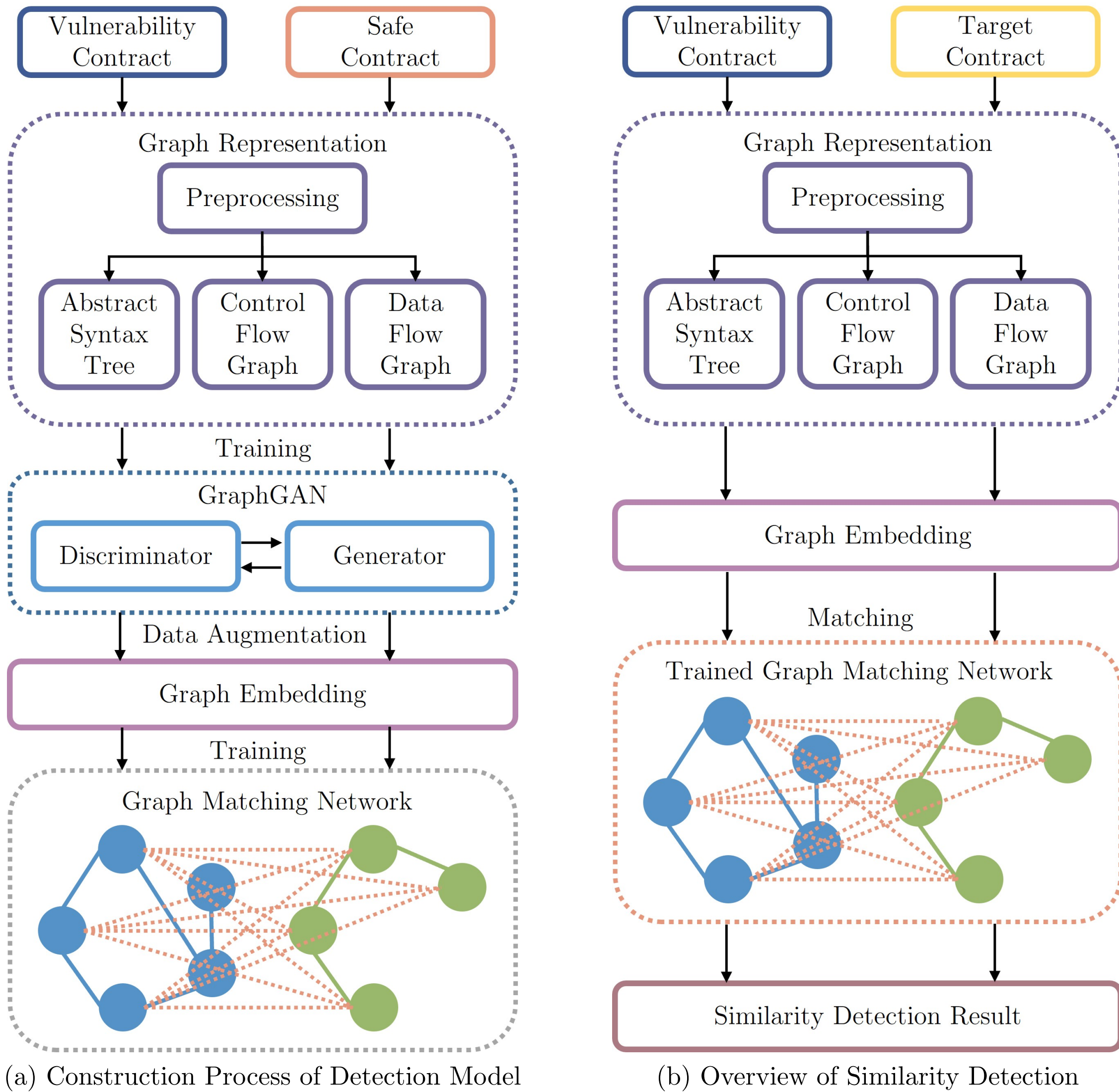
The source code structure of smart contracts is more complex than traditional two-dimensional vector data or text serialization data, and they exhibit non-Euclidean space characteristics. Inspired by the vulnerability detection method based on bytecode matching in Huang's research [17], we explore the vectorization and graph representation of smart contract source code. In our method, the Solidity code is converted into Contract Graphs (CG) that contain rich semantic and structural information, where the basic elements and statements of smart contracts are regarded as nodes, and their relationships constitute edges. We are committed to building an effective source code representation method that preserves its semantic and structural features, providing a solid foundation for identifying and analyzing vulnerability features.

GMN, as an extended form of graph neural network [90], calculates the graph feature similarity between the target and vulnerable contract through the cross-graph attention mechanism. This method maximizes the capture and learning of smart contract structure and semantic information, thereby showing higher accuracy and efficiency in detecting reentrancy and integer overflow. However, GMN's demand for large amounts of data is still a major obstacle. In practical applications, obtaining sufficient and valid data is often infeasible. For this reason, we continue to use the GAN to solve the data shortage problem, thus providing necessary support

for building an efficient GMN model. Specifically, we use graph-based GAN [104] to expand the small sample training set into a large sample training set, which is used to train the GMN-based detection model. The trained GMN model will be used to judge the feature similarity between the target and vulnerable contracts. The experimental results show that our model has high accuracy efficiency, and scalability in detecting reentrancy and integer overflow.

4.2 Methodology Framework

This section details a novel and effective detection method for reentrancy and integer overflow vulnerabilities in Ethereum smart contracts. The high-level idea is to use graphs to represent smart contract source codes, and innovatively use GAN to augment the training set, which is used to train GMN to generate a detection model. Using the trained GMN model to perform similarity detection on smart contracts, and determine whether the contract contains reentrancy or integer overflow vulnerabilities. Figure 4.1 is a framework of the smart contract detection model training process and similarity detection.



(a) Construction Process of Detection Model (b) Overview of Similarity Detection

Figure 4.1: Framework for Detection Method Based on Generative Adversarial Networks and Graph Matching Networks.

To build the detection model, we need to label and classify all the smart contracts in the training set according to their security attributes. Smart contracts are divided into three categories: safe contracts, integer overflow contracts and reentrancy contracts. The detection model construction process is shown in Figure 4.1a. For a given smart contract in the training set, whether it is safe or vulnerable, irrelevant fragments are first removed. We extract the Abstract Syntax Tree (AST) [102], Control Flow Graph (CFG) and Data Flow Graph (DFG) from its Solidity code and integrate them into an undirected multigraph, which can preserve the semantic and structural features of the contract as much as possible. The preprocessed smart contracts will be transformed into CGs, which will be used as the training set for the data augmentation model. Then, we construct a GraphGAN model consisting of a discriminator and a generator. The generator is responsible for generating neighboring nodes, which are evaluated by the discriminator to determine their authenticity. The small-sample training set obtained from the previous process will be used for the adversarial training of the discriminator and the generator. When GraphGAN is trained, the generator will generate a large number of synthetic graphs, which will be embedded as graph vectors. The graph vectors will be paired as the input of the GMN. We will adjust the parameters in the model according to the specified loss function, until the model fully learns the features of the vulnerability.

The trained GMN model uses a cross-graph attention mechanism to calculate the similarity between the target and vulnerable contracts. The detection process for each vulnerability type is the same, and the specific detection procedure is illustrated in Figure 4.1b. The contract is converted into a graph structure data that captures its structure and semantics, before detecting a specific type of vulnerability. A GMN model, trained for detecting that type of vulnerability, takes as input a pair of graph structure data: one from the contract, and another from a randomly selected contract with the same type of vulnerability from a training set. The model outputs two graph vectors, which represent the features of the two contracts. The cosine similarity function measures the similarity coefficient between the two graph vectors, and compares it with the preset similarity threshold coefficient. If the similarity coefficient is higher than the threshold, the contract is detected to have the vulnerability; otherwise, a new pair of graph structure data is formed with another contract from the training set of that type of vulnerability, and the process is repeated. The contract is detected to have no vulnerability, if none of the pairs of graph structure data has a similarity coefficient higher than the threshold.

4.3 Graph Representation of Smart Contract

A common challenge in vulnerability detection using deep learning is the reliance on a single method to represent code, such as AST, CFG, and text tokens. To address the data representation obstacle, we explore and design a preprocessing method to obtain the graph representation of the Solidity code. Figure 4.2 illustrates the overview of graph representation, which consists of two stages: source code preprocessing and construction of CG.

4.3.1 Source Code Preprocessing

In the source code preprocessing part, we face a critical ethical challenge: how to effectively construct the dataset required for the detection model while protecting personal privacy and sensitive information. The source code of smart contracts, especially those written in Solidity, often contains sensitive data such as transaction records and user identity information. Using these data directly for model training may violate data privacy protection regulations. As dis-

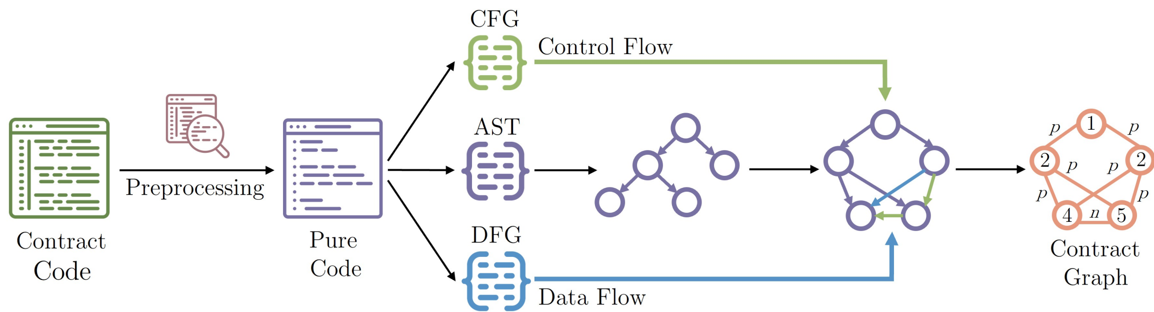


Figure 4.2: Process of Graph Representation.

cussed in Section 1.2, we collect secure open-source data for model training and testing. To further avoid data ethical risks, we apply strict anonymization and de-identification measures to the collected data before preprocessing. More specifically, the method implements a comprehensive data de-identification strategy, which thoroughly anonymizes all data that can be traced back to personal identities, for example, deleting or replacing user identity, contact information, address, and transaction hash. Moreover, transaction information that is irrelevant to vulnerability features, such as transaction time, participants, and purpose, is also removed. These ethical measures not only protect the rights and interests of data subjects, but also ensure that our research activities comply with legal and regulatory requirements.

In order to represent smart contracts as graphs, we need to preprocess the solidity code to eliminate the impact of different developers' coding styles on the graph representation. Coding styles include naming conventions and programming habits for identifiers such as function names and variable names. If preprocessing is not done, code snippets with the same structure and semantics may be converted into different graphs, which will affect the training performance of models and the performance of similarity detection. Therefore, we need to apply the following rules for preprocessing: delete useless fragments in the source code, such as comments, spaces and blank lines, etc.; retain the key features in the source code, such as statements and expressions related to vulnerabilities, etc. Through such preprocessing, we can reduce the noise of the data samples and improve the efficiency of preprocessing and model training.

Moreover, we need to clarify the causes and key features of the vulnerability. The integer overflow is described and analyzed in Section 3.3.1, so this section will summarize the key information of the reentrancy vulnerability. The reentrancy is caused by the contract not properly updating its state variables before the external call, allowing the target of the external call to re-enter the contract's internals and perform unexpected operations. The root cause of the reentrancy is the callback mechanism of Solidity smart contracts. That is, when the contract receives a transfer, it automatically triggers the $fallback$ function. If the $fallback$ function contains a call to the source contract of the transfer, it may form a recursive call, which affects the normal logic of the target contract. Table 4.1 shows some key features of vulnerabilities. Manual review of reentrancy in the source code includes but is not limited to the following two points: check the function calls in the contract, such as $call.value()$ or $transfer()$ etc.; pay attention to the calls to other contracts in the contract, especially when the called contract can be maliciously controlled.

Table 4.1: Examples of Vulnerability Feature.

Vulnerability	Key Features			
Integer Overflow	*add*	*mul*	···	*sub*
Reentrancy	*fallback*	*call.value*	···	*returns (bool)*

4.3.2 Construction of Contract Graph

In Chapter 3, we demonstrate the contribution of AST in smart contract feature extraction. This section will continue to propose and implement a deeper level of contract representation based on ASTs. More specifically, we combine AST, CFG and DFG to represent smart contract source code as graphs, which can better preserve the semantic and structural features.

To capture the syntactic structure of the code, we use solidity-parser [115] to generate the AST of the code. AST offers an abstract syntactic structure of smart contracts, in which leaf nodes correspond to operands and non-leaf nodes signify operators or various other syntactic elements. To differentiate various node types within the AST, unique identifiers will be assigned to each node type. Through these identifiers, we can construct context paths that represent semantic information of the code. These paths represent different semantic fragments of code execution. Figure 4.3 shows the AST of a smart contract RE containing a reentrancy vulnerability.

Listing 4.1: An Example of Smart Contract with Reentrancy

```
1  contract RE {
2      mapping (address => uint256) public wallet;
3
4      function withdraw(uint256 amount) public {
5          if (amount > 2023 && wallet[msg.sender] >= amount) {
6              msg.sender.transfer(amount);
7              wallet[msg.sender] -= amount;
8          }
9      }
10 }
```

Table 4.2: Control Structures of Solidity Code.

Control Structures				
if-else	*if-else if*	*do-while*	...	*assembly*
if	*while*	*for*	...	*break*
continue	*require*	*revert*	...	*return*

It should be noted that AST, as a fine-grained syntactic representation, has a considerable computational overhead that cannot be ignored. If the code has a significant size, AST will be extremely complex, leading to problems such as gradient explosion or overfitting in subsequent model training. Therefore, preprocessing needs to introduce coarse-grained syntax to solve this problem. We use the structure of AST and the control structures in Solidity to construct

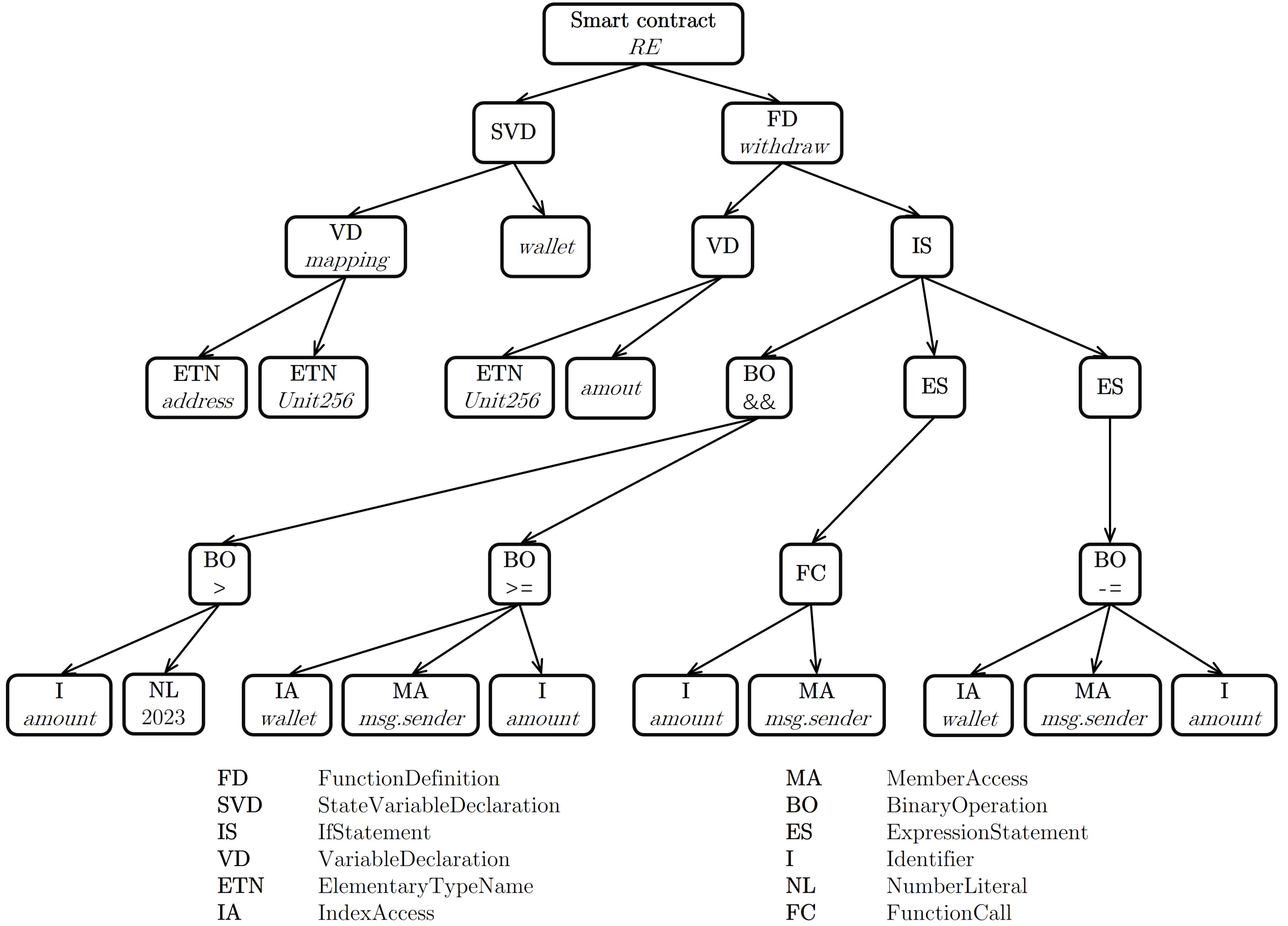


Figure 4.3: Abstract Syntax Tree of Smart Contract RE.

the CFG of smart contracts. CFG can show the data dependency, control dependency and sequential relationship between different statements in the source code. Table 4.2 lists the control statements used to generate CFG. Figure 4.4 shows the CFG of smart contract RE, where each node corresponds to a statement in the source code. After constructing the CFG, we obtain the data variables and corresponding statements by traversing the code multiple times. More specifically, the data flow direction is used to define the edges between the corresponding statements. As shown in Figure 4.5, we use the above rules to construct the DFG of smart contract RE, which provides the data features that are lacking in AST and CFG.

Consequently, we innovatively designed a statement-level granularity representation form - CG, which consists of AST, CFG, and DFG. We construct the CG based on the structure hierarchy and execution order of AST, and determine the node priority and unique numeric label for the key statements. The nodes in CG correspond to the statements in the source code, and the edges between nodes represent the control relationship and data relationship between statements. The control relationship can be obtained from CFG, which describes the control flow between statements. The control flow is usually controlled by keywords such as if, $while$, for and $break$, etc. If there is a control flow between two nodes, an edge with annotation needs to be added between the nodes according to its type and direction. The data relationship can be obtained from DFG, which describes the data flow between statements. The data flow is also used to add edges between nodes. It should be noted that the data flow depends on the execution order of the statements. In addition, the variables defined in the code need to be

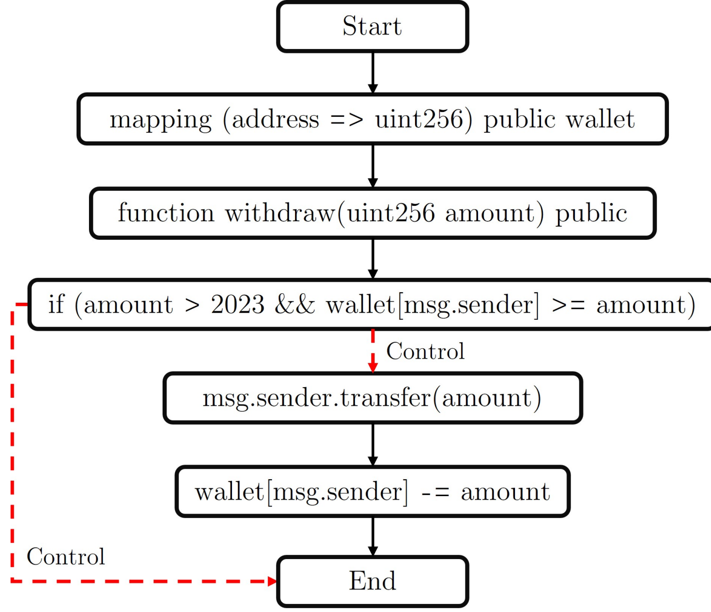


Figure 4.4: Control Flow Graph of Smart Contract RE.

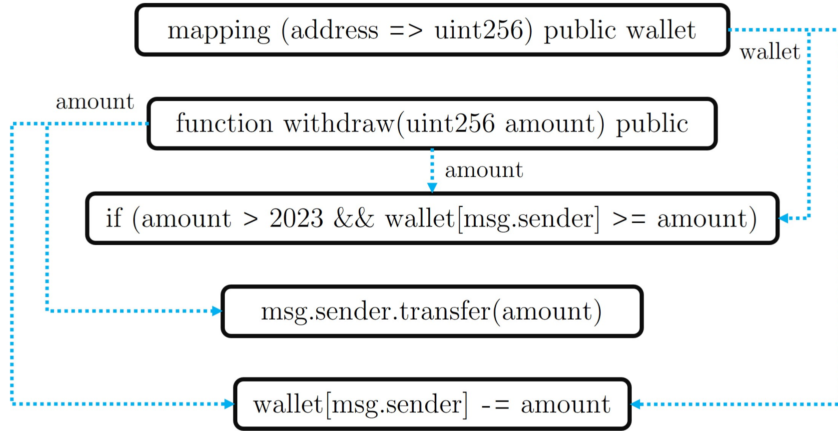


Figure 4.5: Data Flow Graph of Smart Contract RE.

recorded in the annotation of the edge. We first determine the control path in the contract, and then determine the data path by checking the variables on the control path. Finally, the original CG is composed of nodes containing statement information and edges containing control/data information. Figure 4.6 shows the original CG of smart contract RE.

In order to perform data augmentation on the CG, we define the CG as follows:

$$CG = (N, E), \tag{4.1}$$

where N is a set of nodes, and E is a set of edges. Each node in N represents a statement in the contract code. E is composed of the set of undirected edges between each pair of nodes in the graph. The number of edges between the nodes represents the number of control relations and data relations between them. The construction of CG needs to follow the input format of the GraphGAN [25]. We assigned corresponding priority numerical labels to the key nodes according to the top-bottom order of the hierarchy in the AST. Then, we define the direction of control flow and data flow based on the node's priority. Specifically, the path from a high-priority node to a low-priority node is the positive direction (label p); the path from

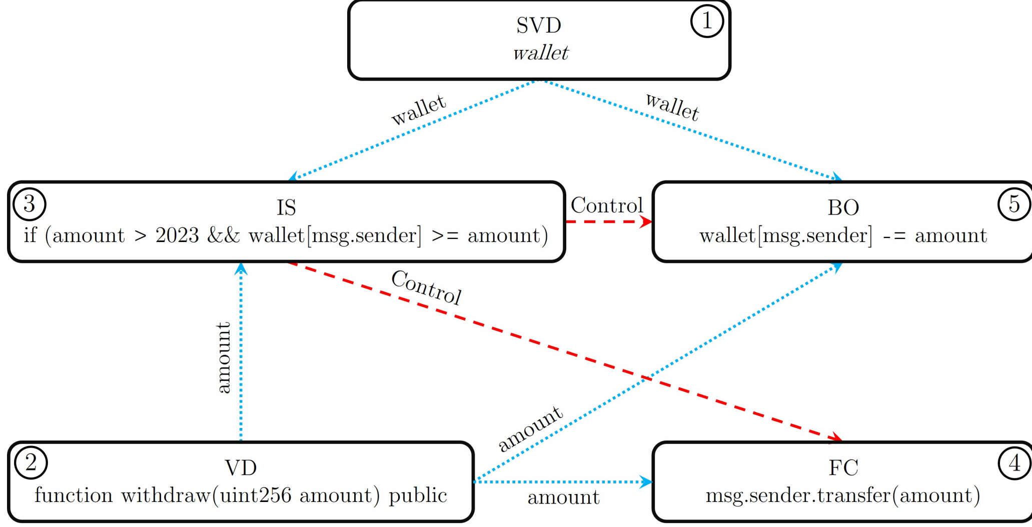


Figure 4.6: Original Contract Graph of Smart Contract RE.

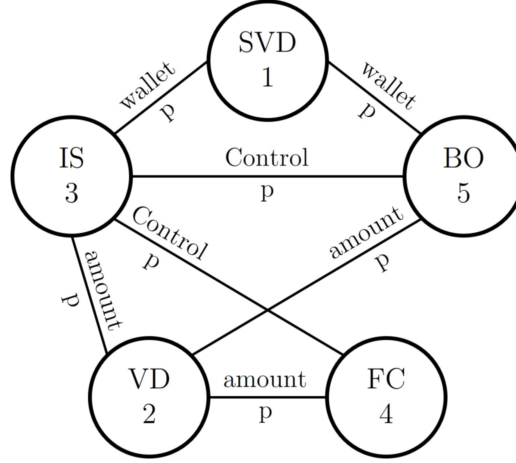


Figure 4.7: Contract Graph (CG) of Smart Contract RE.

a low-priority node to a high-priority node is the negative direction (label n); if there is no path between two nodes, it is the no direction, and the number of undirected edges is 0 (no label). The nodes and edges will be added with label and statement information annotations. The key nodes and edges together form the CG. Figure 4.7 shows the CG of smart contract RE. Compared with the research methods that only use AST or CFG [17] to represent smart contracts, CG is more conducive to GMN extracting the key features of contracts.

4.4 GAN-based Graph Data Augmentation

Figure 4.8 illustrates the running process of dataset augmentation. This detection method innovatively introduces the GraphGAN [25] model to solve the data starvation problem, which is recognized as a hindrance to building Graph Neural Network models. GraphGAN consists of a generator and a discriminator. In the training process of this model, the generator is used to generate neighbor nodes of CG. In contrast, the discriminator calculates the state of the edge between two nodes. More specifically, $CG = (N, E)$ is input to the model, where

$$N = \{n_1, \cdots, n_N\}, \tag{4.2}$$

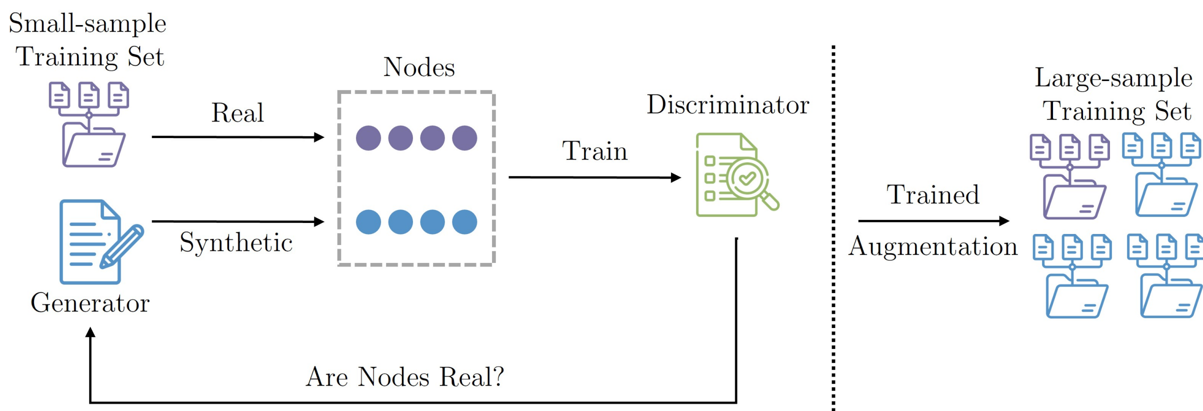


Figure 4.8: Process of Graph Data Augmentation.

represents the set of nodes, and

$$E = \{e_{xy}\}_{x,y \in N}, \tag{4.3}$$

represents the set of edges, e_{xy} represents the set of edges between node x and node y. For a given node n_c, we define $N(n_c)$ as the set of nodes directly connected to n_c, and $N(n_c)$ is always much smaller than N in general. We represent the potential true connectivity distribution of n_c as conditional probability $p_{true}(n|n_c)$, which reflects the connection preference distribution of n_c over all other nodes in N. $N(n_c)$ can be regarded as a set of observation samples drawn from $p_{true}(n|n_c)$.

Based on this, the generator can be defined as $G(n|n_c; \theta_G)$, which tries to approximate the potential true connectivity distribution $p_{true}(n|n_c)$, and generate or select the most likely nodes to connect with n_c. The discriminator can be defined as $D(n|n_c; \theta_D)$, which aims to distinguish the connectivity of node pairs (n, n_c), and output a single scalar, indicating the probability of edge existence between n and n_c, and predicts the number of undirected edges. θ_G and θ_D are the parameters of the GraphGAN, which control the behavior of the generator and the discriminator, respectively. They can automatically learn and adjust through the backpropagation algorithm, enabling GraphGAN to generate high-quality data for various application scenarios. In the training process, the generator gradually learns how to generate synthetic nodes that match the real neighbor nodes, while the discriminator also gradually learns how to evaluate the edge probability between two nodes correctly [25].

The generator $G(n|n_c; \theta_G)$ and discriminator $D(n|n_c; \theta_D)$ are playing a minimax game with the value function $V(G, D)$:

$$\min_{\theta_G} \max_{\theta_D} V(G, D) = \sum_{c=1}^{N} (\mathbb{E}_{n \sim p_{true}(\cdot|n_c)} \log D + \mathbb{E}_{n \sim G(\cdot|n_c;\theta_G)}[\log(1 - D)]), \tag{4.4}$$

where $\mathbb{E}$ is the expectation, which represents the average output value of the discriminator for all possible n under the real distribution $p_{true}(\cdot|n_c)$ or the generator's distribution $G(\cdot|n_c; \theta_G)$ given n_c. The optimal parameters of the generator and discriminator can be learned by alternately maximizing and minimizing the value function $V(G, D)$. When the generator and discriminator reach Nash equilibrium, we can consider that GraphGAN has completed training. In the case of completing training, the model will be able to generate synthetic CGs that are similar to real CGs. In this way, we can generate a large number of synthetic CGs to enrich the training set of GMN.

It should be emphasized that we augment the datasets of safe contracts, integer overflow contracts and reentrancy contracts separately by independently training three GraphGAN models. Independent training aims to reduce the similarity between these three types of synthetic contracts, thereby reducing the false positive detection rate. Each GraphGAN model is trained with the same parameters, which can achieve the effect of fast transfer learning and improve the training efficiency.

4.5 GMN-based Similarity Detection

This section uses a Graph Embedding Network to vectorize the CG containing structural and semantic features. Subsequently, we utilize a GMN to extract vulnerability features from the vectorized CG. This is followed by devising a practical similarity detection process aimed at identifying specific vulnerabilities present in the smart contract source code.

4.5.1 Graph Embedding

To better learn the semantic and structural features of smart contract source code, we need to embed the CG. After obtaining the CG containing key information and implicit vulnerability features of smart contracts, we need to use an embedding model to vectorize the textual information in CG for subsequent feature learning using GMN. More specifically, each node in CG contains an annotation of a source code statement and priority numeric label, and each edge also contains an annotation of a control/data statement and direction label. Since GMN is difficult to handle text-form annotations, we need to use feature vectors to represent the node annotation and edge annotation in CG. In this way, we can better capture the semantic and structural features of smart contract source code, and improve the accuracy and efficiency of vulnerability detection.

The word2vec [103] model is a commonly used word embedding model in the field of Natural Language Processing, whose main idea is to use neural networks to predict the vector distribution of each word. In order to convert the node annotation and edge annotation in CG into dense vector representations, we adopt the Skip-gram model [103] for word embedding. First, we use the text of all nodes in AST as the corpus of Skip-gram. Then, we perform one-hot encoding [103] on the text of each node, resulting in sparse vectors. Next, we use the Skip-gram model and a multi-classifier based on Hierarchical Softmax to train the model.

As shown in Figure 4.9, the core idea of the Skip-gram model is to use the current word to predict the vector representation of the context words, which consists of an input layer, a hidden layer and an output layer. For the input layer, we set a fixed-size window and use the one-hot vectorized source code statements as input. For the hidden layer, we sum and average all the sparse vectors to obtain the hidden vector. After learning from the Skip-gram model, we can construct a vector mapping table of all the text of the statements, thus achieving the vectorization of the annotation of each node and edge in the corpus. Through the Skip-gram model, we can generate the node annotation vector and edge annotation vector of the source code. Finally, all text information in CG can be replaced with vectors.

4.5.2 Similarity Detection Model

GMN is used to build a similarity detection model. Unlike traditional GNNs, GMN does not map each graph to a feature vector independently, but obtains a pair of graphs simultaneously

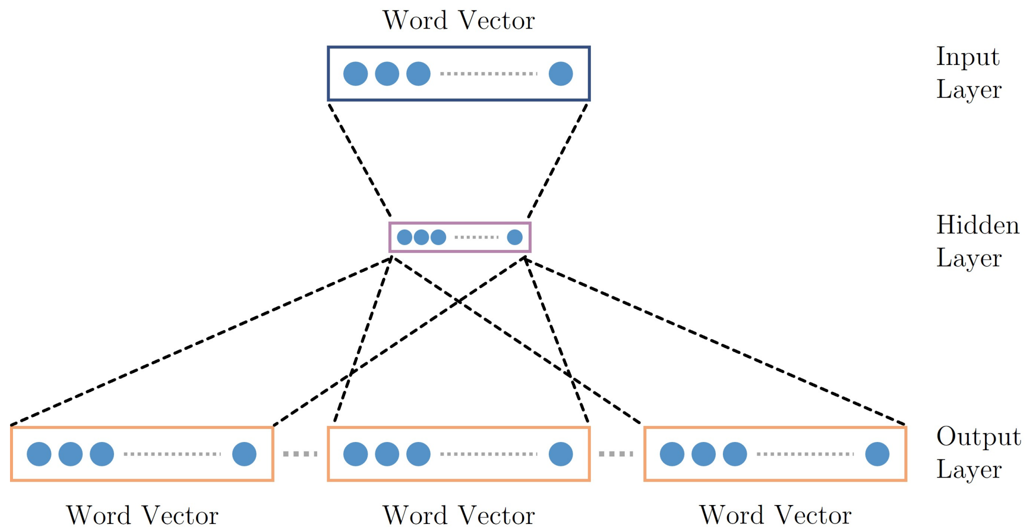


Figure 4.9: Overview of the Skip-Gram Model.

and uses the cross-graph attention mechanism to learn embeddings [19]. By associating the nodes in the graph pair and identifying the differences, GMN can provide a good accuracy-computation trade-off, which is unavailable in traditional embedding models.

Taking the construction of the integer overflow detection model as an example, we need to label the types of safe contracts and integer overflow contracts output by the Dataset Augmentation stage. All contracts will be embedded as graph vectors by word2vec [103] and form the training set. Moreover, the contracts in the training set will be randomly combined into graph pairs. GMN model will use this dataset for training and detect whether the target contract is an integer overflow contract. Similarly, the reentrancy vulnerability detection model also needs to go through the same process for construction. The construction and similarity detection process of the GMN model is shown in Figure 4.10.

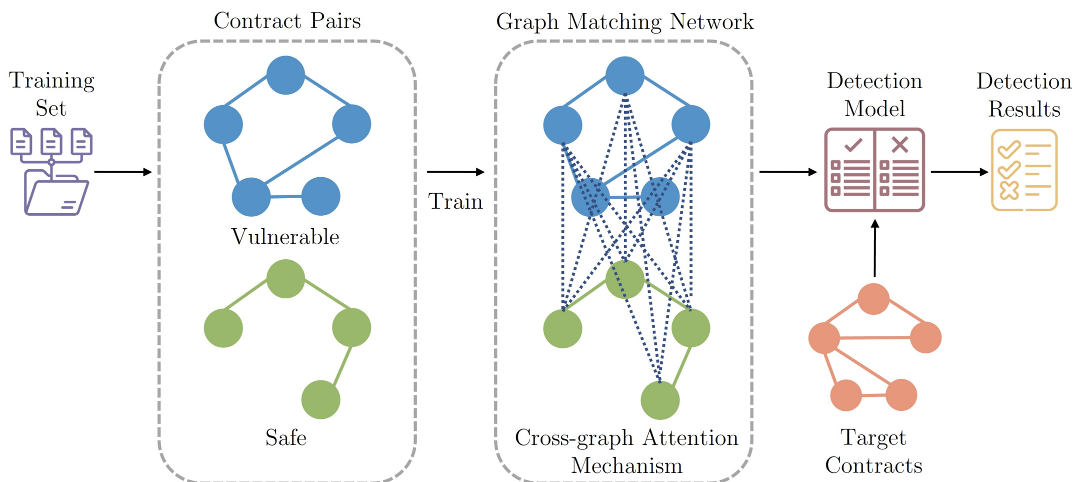


Figure 4.10: Construction and Workflow of Detection Model.

GMN model consists of an encoding layer, a propagation layer and an aggregation layer. The encoding layer will use Multi-Layer Perceptron (MLP) [22] to extract and encode the features

of nodes and edges in $CG = (N, E)$:

$$\eta_i^{(0)} = MLP_{node}(\alpha_i), \forall i \in N; \tag{4.5}$$

$$e_{ij} = MLP_{edge}(\alpha_{ij}), \forall (i,j) \in E, \tag{4.6}$$

where α_i is the feature vector of node n_i, α_{ij} is the feature vector of edge e_{ij}, $\eta_i^{(0)}$ is the initial embedding of the ith node, and e_{ij} is the initial embedding of the edge between the ith and the jth nodes.

The propagation layer introduces the attention mechanism, which can perform information interaction between two graphs, so as to better capture the structural and semantic similarity between graphs [19]. The propagation layer in the graph network consists of three steps, which update the edge vector, node vector and graph vector in the graph network respectively:

- When updating the edge vector, we aggregate the initial vector attribute of the edge to be updated with the vector of all adjacent nodes, and then replace the original edge vector.
- When updating the node vector, we need to aggregate the initial vector attribute of the node, the vector of all outgoing edges and incoming edges adjacent to the node, as the input of the node update function, and obtain the new node vector.
- When updating the graph vector, we need to use the initial vector attribute of the graph, the vector of all nodes and edges in the graph as the input of the graph vector update function, and obtain the updated graph vector after calculation.

Compared with ordinary GNNs, GMNs have stronger discriminative and expressive abilities, and can distinguish some graph structures that GNNs cannot distinguish. Given two contract graph $CG_1(N_1, E_1)$ and $CG_2(N_2, E_2)$, at each iteration t, the propagation layer formula is as follows:

$$\mu_{j \to i} = f_m(\eta_i^{(t)}, \eta_j^{(t)}, e_{ij}), \forall (i,j) \in E_1 \cup E_2; \tag{4.7}$$

$$\nu_{j \to i} = f_c(\eta_i^{(t)}, \eta_j^{(t)}), \forall i \in N_1, j \in N_2 \; or \; i \in N_2, j \in N_1; \tag{4.8}$$

$$\eta_i^{(t+1)} = f_u(\eta_i^{(t)}, \sum_j \mu_{j \to i}, \sum_{j'} \nu_{j' \to i}), \tag{4.9}$$

where $\eta_i^{(t)}$ and $\eta_j^{(t)}$ are respective sets of node hidden vectors for CG_1 and CG_2; f_m is the message function that collects messages from the neighborhood of node n_i, f_c is used for cross-graph information computation. The function f_u serves as the node update mechanism, integrating these messages with the ultimate hidden state of the node embedding. As shown in Figure 4.11, the attention formula between node i in CG1 and any node in CG2 is as follows:

$$a_{j \to i} = exp(s_\eta(\eta_i^{(t)}, \eta_j^{(t)})) / \sum\nolimits_{j'} exp(s_\eta(\eta_i^{(t)}, \eta_j^{\prime(t)})), \tag{4.10}$$

and therefore,

$$\sum_j \nu_{j \to i} = \sum_j a_{j \to i}(\eta_i^{(t)} - \eta_j^{(t)}) = \eta_i^{(t)} - \sum_j a_{j \to i}\eta_j^{(t)}, \tag{4.11}$$

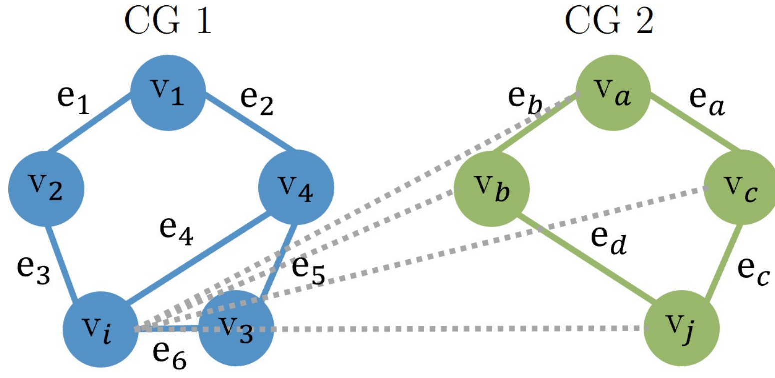


Figure 4.11: Example of Cross-graph Attention Mechanism.

where s_η is a vector space similarity measure, $a_{j\to i}$ are the attention weight, and $\sum_j \nu_{j\to i}$ intuitively measures the difference between $\eta_i^{(t)}$ and its nearest neighbour $\eta_j^{(t)}$ in another graph.

After T rounds of updates in the graph, it is necessary to generate a vector representing the global graph structure information by aggregating the information of all nodes in the graph:

$$\eta_{CG_1} = f_a(\{\eta_i^{(T)}\}_{i\in N_1}); \tag{4.12}$$

$$\eta_{CG_2} = f_a(\{\eta_i^{(T)}\}_{i\in N_2}), \tag{4.13}$$

where f_a is the graph aggregation function.

The following formula can calculate the graph pair similarity in the aggregation layer:

$$s = f_s(\eta_{CG_1}, \eta_{CG_2}), \tag{4.14}$$

where s is the similarity score ($s \in [-1, 1]$), f_s is the cosine similarity between η_{CG_1} and η_{CG_2}. The closer s is to -1, the more dissimilar the two graphs are; the closer s is to 1, the more similar the two graphs are.

To detect different types of smart contract vulnerabilities, we use the same process to train GMN models for various vulnerabilities. Taking reentrancy vulnerability as an example, our process is as follows:

1) We label each smart contract in the training set as Reentrancy or Safe, indicating its security.
2) According to the input format and cross-graph attention mechanism of GMN, we randomly pair the CG in the training set as input data. If the labels in the graph pair are consistent, we mark them as Positive, and if the labels in the graph pair are inconsistent, we mark them as Negative. Therefore, all graph pairs are divided into two sets: Positive and Negative.
3) We use the detection model to analyze all graph pairs, compare the similarity score s with the set (Positive or Negative) to which the graph pair belongs, and update the parameters in GMN using the backpropagation algorithm and loss function, so that the model can learn the vulnerability features of smart contracts.

As shown in Figure 4.1b, we convert the target smart contract into the corresponding CG. The target CG is paired with the CG in the target vulnerability set, and input to the GMN model to calculate the similarity score, and then compared with the preset threshold to determine whether the target contract has a target vulnerability. We will evaluate and analyze the effectiveness and scalability of this detection method through experiments in Section 4.6.

4.6 Experimental Result and Analysis

This section aims to verify the effectiveness, advantages, and scalability of the proposed method in detecting integer overflow and reentrancy vulnerabilities of smart contracts through experiments. In the previous sections, we introduce how to transform the smart contract source code into CGs containing structural and semantic features, how to use GAN for data augmentation to expand the training set, and how to embed CG and construct GMN to determine whether smart contracts have vulnerabilities. This section will first describe the experimental environment and dataset, then determine the optimal threshold coefficient of the GMN model through experiments, compare and analyze with other related methods, and finally conduct experimental analyses on the scalability of the proposed method.

4.6.1 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 use solidity-parser-antlr [102] to generate the AST from the smart contract source code, and use word2vec [103] to extract the feature vectors of node and edge from the CG. To evaluate the effectiveness and advantages of the proposed method in detecting integer overflow and reentrancy vulnerabilities in smart contract source code, we designed the following two experimental steps:

1) Graph Matching Network parameter selection: In Section 4.6, we proposed a smart contract vulnerability method based on the GMN model, and the parameter selection of the GMN model will directly affect the model's learning effect of smart contract vulnerability features. We will explore the learning performance of the GMN from five aspects - embedding dimension, hidden layer number, learning rate, iteration number and similarity threshold. Through experimental comparison, we select the optimal parameters to make the GMN's vulnerability detection ability reach the optimal state.

2) Vulnerability detection effectiveness evaluation: We collected smart contracts with security labels from the open-source platform, formed our base dataset, and divided it into training and testing sets. Then, we compared the proposed method with four existing detection tools analyzed in Section 2.4 to demonstrate the effectiveness and advantages of our method.

3) Method scalability evaluation: We evaluate the proposed method's computational efficiency, data requirement, and testing efficiency by constructing specialized training sets and conducting experiments with incremental data samples. The experimental results aim to reveal the scalability of the method in handling large-sample smart contracts and provide insights for model optimization.

4.6.2 Dataset and Evaluation Metric

Dataset

To evaluate the effectiveness and advantages of the proposed method in detecting reentrancy and integer overflow vulnerabilities of smart contract source code, we collected 550 smart contracts from the Etherscan platform [112] as our dataset. These smart contracts all contain solidity source code, contract address and security attribute labels, for us to conduct further analysis and evaluation. We divided these smart contracts into two datasets, as shown in Table 4.3:

- The training set contains 50 safe contracts, 100 reentrancy contracts, and 100 integer overflow contracts. This training set is used to train the GAN model for data augmentation.
- The test set contains 300 smart contracts, including 100 safe contracts, 100 reentrancy contracts, and 100 integer overflow contracts. This test set is used to evaluate the detection performance of our method and other tools.

Table 4.3: Summary of Original Datasets.

Type of Dataset	Number of Smart Contracts		
	Safe	Reentrancy	Integer Overflow
Training Set	50	100	100
Test Set	100	100	100

In addition, we also use the trained GAN model to augment the number of safe, reentrancy, and integer overflow contracts in the original training set to 500, 1000, and 1000, respectively. The augmented training set is used to achieve the aim of the GMN model's few-shot learning, as shown in Table 4.4.

Table 4.4: Summary of Augmented Datasets.

Type of Dataset	Number of Smart Contracts		
	Safe	Reentrancy	Integer Overflow
Original Training Set	50	100	100
Augmented Training Set	500	1000	1000

To further evaluate the scalability of the proposed method, we select 200 labeled smart contracts from the open-source dataset SmartBugs [12] as the test set for the experiments in Section 4.6.6.

Evaluation Metric

After preparing datasets, we selected four vulnerability detection tools sFuzz [11], Oyente [10], Securify [9] and CGE [18] to conduct the experimental comparison. The selection of these four tools is based on the following principles:

- The source code of the detection tool is publicly available.

- Many vulnerability detection tools have chosen this tool as the testing benchmark when testing their own effectiveness.
- This tool can detect the type of vulnerability that we are concerned with.

We used these four tools to perform vulnerability detection on the test set. Subsequently, we analyzed, evaluated, and compared the detection results to demonstrate the effectiveness and advantages of the vulnerability detection method proposed in this chapter. When testing the performance of each tool, we used Confusion Matrices [113] to represent the detection results. Based on this, Accuracy(ACC) [113] is used as the evaluation metric for the detection model:

$$ACC = \frac{TP + TN}{TP + TN + FP + FN}. \tag{4.15}$$

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 ACC, the higher the precision of the model.

4.6.3 GMN Parameters Experiment

The parameters of the GMN model will affect the detection performance of smart contracts, so this section will explore the optimal values of the parameters through experiments. More specifically, we will adjust the similarity threshold, number of hidden layers, iteration number, learning rate and embedding dimension, respectively, and observe their impact on the detection performance, so as to select the most suitable parameter combination for integer overflow and reentrancy vulnerability detection, and make the proposed method achieve the best performance in detection.

Embedding Dimension

The embedding dimension represents the size of the node and edge feature vectors in the graph. This parameter directly affects the computational complexity and feature capture ability of the model. A larger embedding dimension can capture more information from the CG, but it will increase the computational complexity, while a smaller embedding dimension can reduce the computational complexity, but it may miss some key features.

Figure 4.12 shows the impact of the embedding dimension on the model detection performance:

- Starting from 40, when the embedding dimension is low, the detection accuracy increases sharply with the increase of the embedding dimension, until the embedding dimension reaches 100, the detection accuracy of reentrancy and integer overflow reaches the maximum value.
- After the accuracy reaches the peak, the detection performance decreases continuously with the increase of the embedding dimension.

According to the experimental results, it is obvious that when the embedding dimension is low, the feature vectors generated by the GMN model lack some key features of the CG, resulting in underfitting of the detection results; when the embedding dimension exceeds 100, the detection model overfits, and it should be noted that the learning cost of the model increases with the

increase of the embedding dimension. In comparison, the GMN model finally selects 100 as the parameter of the embedding dimension of the CG.

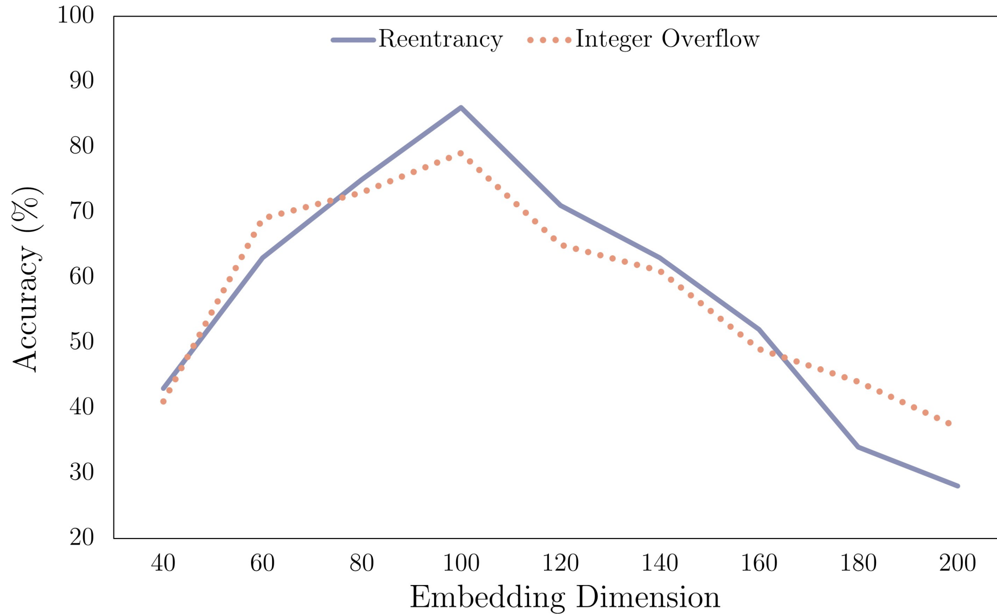


Figure 4.12: Experiments on the Embedding Dimension.

Number of Hidden Layers

The number of hidden layers in the GMN model, representing the propagation layers, plays a pivotal role in the model's performance. Increasing the number of hidden layers can improve the model's linear classification ability for different types of contracts, but it may also cause overfitting. Specifically, a higher count of hidden layers diminishes the GMN model's sensitivity to vulnerability features in the CG, thereby reducing classification effectiveness. Conversely, fewer layers can lower the overfitting risk but might compromise classification ability.

Figure 4.13 shows the impact of the number of hidden layers on the model detection performance:

- The detection accuracy of the model gradually increases as the number of hidden layers increases from 1.
- When the number of hidden layers is 4, the GMN model's detection accuracy for reentrancy and integer overflow reaches the peak.
- When the number of hidden layers exceeds 4, the model detection accuracy drops sharply with the increase of the number of layers

In comparison, the GMN model optimally employs four hidden layers, balancing classification capabilities with the risk of overfitting.

Learning Rate

The learning rate is a key parameter in the GMN model. A large learning rate can lead to oscillations, causing excessive updates in feature parameters and impeding convergence to an

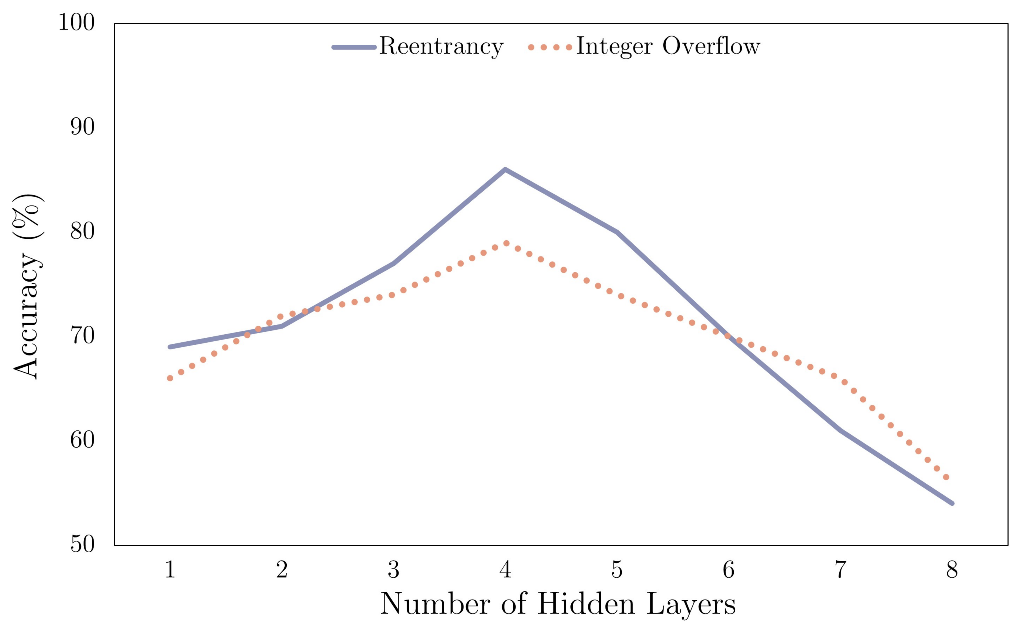


Figure 4.13: Experiments on the Number of Hidden Layers.

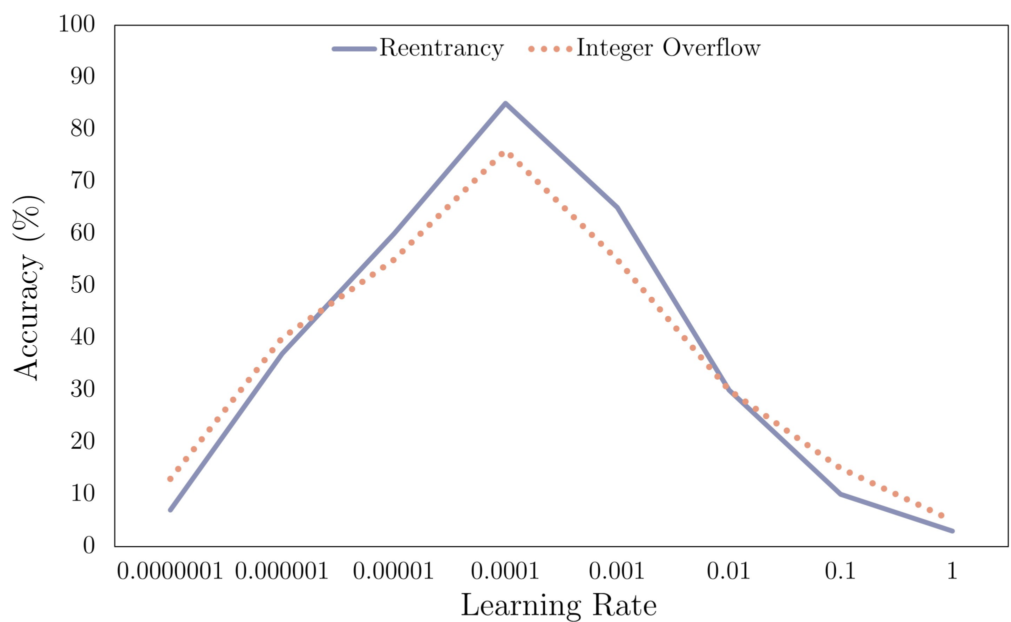


Figure 4.14: Experiments on the Learning Rate.

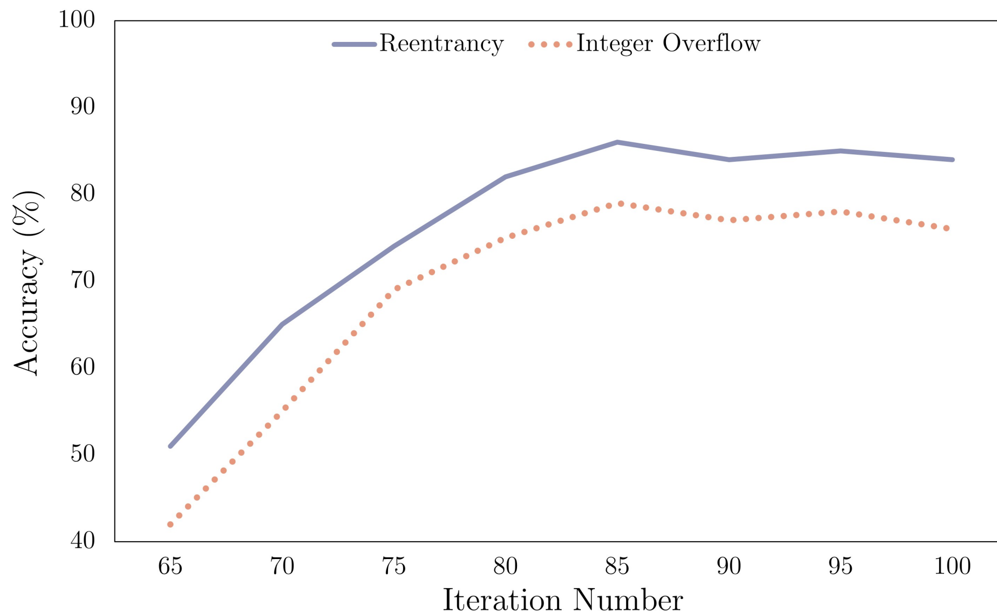


Figure 4.15: Experiments on the Number of Iteration.

optimal solution. Conversely, a small learning rate might result in overly slow convergence. Figure 4.14 shows the impact of the learning rate on the model detection performance:

- The detection accuracy of the GMN model generally improves with an increasing learning rate in the first half of the experiment.
- When the learning rate is 0.0001, the detection accuracy of the model reaches the highest point.
- When the learning rate continues to increase, the accuracy of the model shows a downward trend.

Considering comprehensively, the GMN model finally selects 0.0001 as the number of the learning rate.

Iteration Number

In the GMN model, iteration refers to the number of times all node and edge vectors in the CG are updated. Too many iterations may cause the model to overfit, and too few iterations may cause the model to underfit. Figure 4.15 shows the impact of iteration number on the model detection performance:

- Before reaching the optimal performance, the detection accuracy of the GMN model increases with the increase of the number of iterations.
- When the number of iterations is 85, the detection performance of the model reaches the highest point.
- When the number of iterations continues to increase, the performance of the model tends to be stable, indicating a diminishing return on additional iterations.

To balance model accuracy and computational efficiency, and to mitigate risks of overfitting, the GMN model is configured to undergo 85 iterations for training.

Similarity Threshold

The similarity threshold in the GMN model refers to the decision value for the similarity score s. When s is below this threshold, the model deems the target contract as non-vulnerable; exceeding the threshold indicates potential vulnerabilities. The choice of similarity threshold directly influences the model's sensitivity to similarities in the CG. A higher threshold heightens the model's accuracy requirements for CG similarity, potentially increasing precision. However, excessively high thresholds may lead to increased false positives, while too low thresholds can cause higher false negatives. Furthermore, the threshold selection also impacts the model's complexity; both excessively high and low values can diminish performance.

Figure 4.16 illustrates the impact of similarity threshold on model detection performance:

- Prior to achieving optimal performance, the detection accuracy of the GMN model shows an upward trend with increasing similarity thresholds.
- The peak performance is observed at a threshold of 0.8, beyond which accuracy declines.

Consequently, after a comprehensive evaluation, a similarity threshold of 0.8 is established for the GMN model.

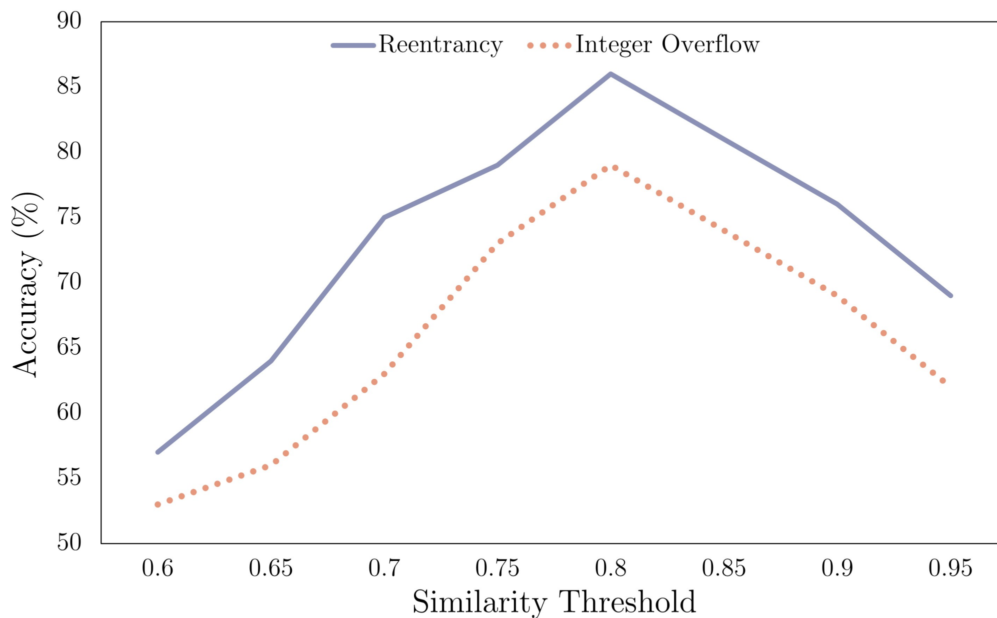


Figure 4.16: Experiments on the Similarity Threshold.

4.6.4 Detection Accuracy Experiment

To evaluate the detection efficacy of the methods proposed in this chapter against other tools, we employed a test dataset comprising 100 safe contracts, 100 integer overflow contracts, and 100 reentrancy contracts. Initially, contracts from the test set were segregated into different folders based on their security types. Subsequently, we ran our model, along with Oyente [10],

sFuzz [11], Securify [9], and CGE [18], to detect vulnerabilities in the contracts within each folder. Additionally, the experiment incorporated a unique control group to analyze the role of GraphGAN [25] in constructing detection models. This experiment group (Model Ω) did not utilize GraphGAN for data augmentation but was trained directly using the original training set. Finally, we compared the output results of each detection model against the original labels of the contracts. This comparison yielded the count of TP, FP, NP, and TN for each model. Based on these counts, we calculated the accuracy rates of the various detection tools.

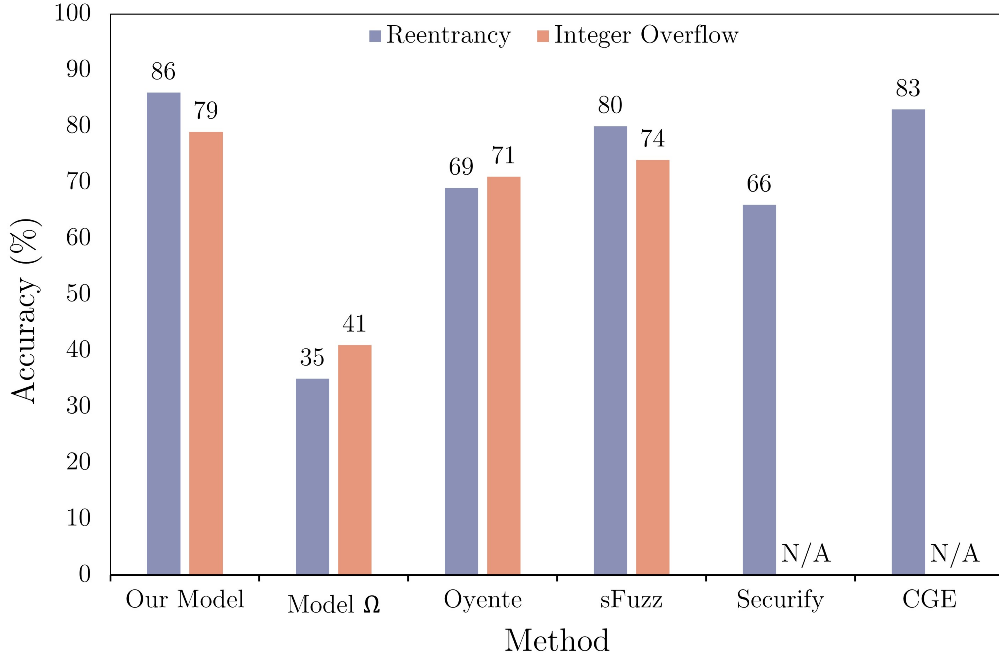


Figure 4.17: Methods Accuracy Experiment.

The experiment result is shown in Figure 4.17. It is known that the original training set has a limited sample size, containing only 250 contract instances (including 50 safe contracts, 100 reentrancy contracts and 100 integer overflow contracts). Compared with the GMN model using GraphGAN for the training set augmentation, Model Ω significantly reduces the accuracy of detecting reentrancy and integer overflow. Specifically, Model Ω has an accuracy of 35% for reentrancy detection and 41% for integer overflow detection, compared with our model, which has an accuracy of 86% and 79%, respectively.

The limitation of sample size leads to Model Ω being more prone to overfitting. Overfitting occurs when the model overlearns the specific features in the training set during the training process, and cannot effectively generalize to new, unseen data. In our case, this means that Model Ω may perform well on the training set, but fail to accurately identify all the vulnerability features on the broader and more diverse test set contracts. Therefore, this finding highlights the importance of using larger and more diverse training datasets, especially in complex and variable contract vulnerability detection scenarios. It also demonstrates the potential value of data augmentation techniques such as GraphGAN in improving the model's generalization ability and accuracy.

In the comparative experiment, our model achieved an accuracy of 86% in detecting reentrancy vulnerabilities and 79% in detecting integer overflow vulnerabilities. This performance is significantly better than other tools. For example, Oyente, which uses symbolic execution, has

an accuracy of 69% and 71%, respectively, for reentrancy and integer overflow; sFuzz, which uses fuzzing, achieves 80% and 74%; Securify, which uses formal verification, has an accuracy of 66% for reentrancy vulnerability detection. It is worth mentioning that CGE, which also uses graph neural networks, has a high accuracy of 83% for reentrancy vulnerability detection, second only to our model.

The outstanding performance of our model is attributed to its unique GAN, which is used for data augmentation, greatly enhancing the model's learning ability when dealing with small sample data sets. Through GAN, our model can generate diverse contract samples, which are synthetic, but very similar to real vulnerable contracts in terms of features, thus providing more rich and diverse training data for GMN. This data augmentation strategy enables GMN to effectively learn and identify complex vulnerability patterns in a small sample learning environment. In addition, the introduction of GMN further improves the model's ability to capture the structural and behavioral features of smart contracts. Therefore, by combining the data augmentation ability of the GAN and the efficient learning mechanism of the GMN, our model shows a significant technical advantage in the field of smart contract security detection, especially in dealing with small sample data sets and identifying complex vulnerability patterns.

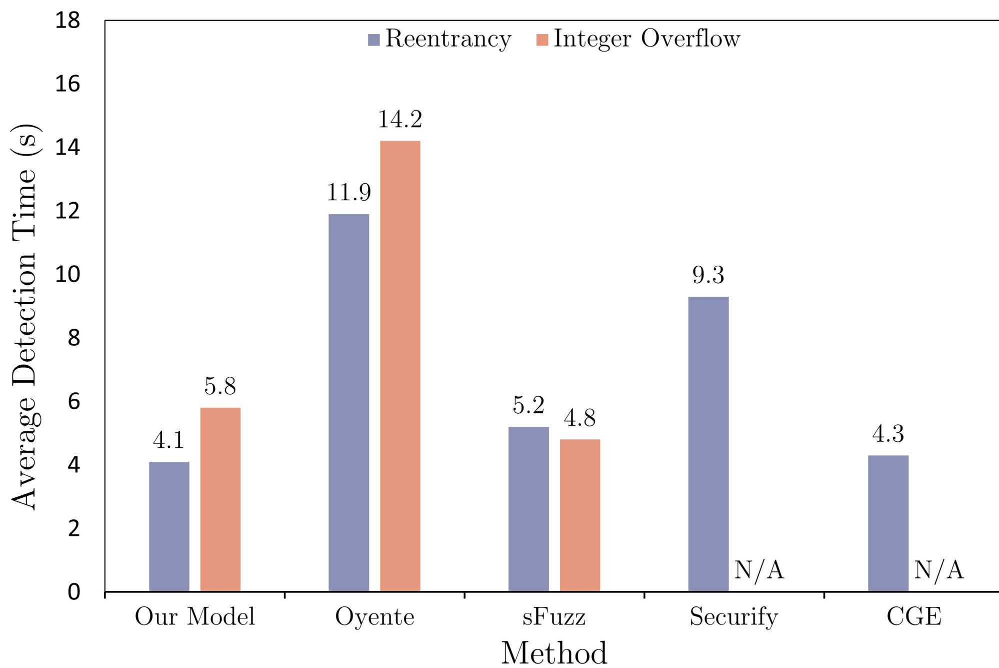


Figure 4.18: Methods Efficiency Experiment.

4.6.5 Detection Efficiency Experiment

We conducted experiments by calculating each contract's average detection time to measure the method's efficiency. As shown in Figure 4.18, our model showed excellent efficiency in detecting integer overflow, with an average detection time of 5.8 seconds. This time is second only to sFuzz (4.8 seconds), and much faster than Oyente (14.2 seconds). In detecting reentrancy vulnerabilities, our model performed even better, with an average detection time of only 4.1 seconds, lower than Oyente (11.9 seconds), sFuzz (5.2 seconds), Securify (9.3 seconds) and CGE (4.3 seconds). These results show that our model achieves high detection speed while ensuring high detection accuracy.

Based on the experimental results, we draw the following analysis conclusions:

- Oyente's core is symbolic execution, which identifies smart contract vulnerabilities by creating and updating CFGs, and using Z3 solver to handle conditional jumps. This method relies on the solver's results to determine the execution paths. However, symbolic execution faces challenges when analyzing contracts that contain complex semantic features such as environmental variables, library functions and dynamic calls. This limitation largely reduces Oyente's efficiency in smart contract vulnerability detection.
- sFuzz applies the fuzzing to detect smart contract vulnerabilities. It tests the robustness of contracts by inputting invalid or randomized data. To achieve this, sFuzz needs to perform a series of preprocessing steps before starting the test, including building CFGs and collecting dependency relationships between contract functions. Then, based on this preprocessing information, sFuzz uses random mutation and coverage-driven strategies to generate test inputs. However, this preparatory work may become inefficient when dealing with contracts with complex semantics, which affects the overall speed and effectiveness of sFuzz in smart contract vulnerability detection.
- Securify uses formal verification, which focuses on verifying the security of smart contracts by rigorous mathematical methods. Although this method can theoretically provide high accuracy, it may be limited by the code complexity and diversity of contracts in practical applications. Especially when dealing with complex smart contracts, formal verification may take more time to complete a comprehensive analysis, which affects its overall detection efficiency.
- CGE uses Graph Convolutional Neural Network (GCN) [116], [117], which is an advanced technique for effectively analyzing graph data structures. GCN is particularly suitable for parsing the dependency and interaction relationships of smart contracts, which gives CGE a significant advantage in understanding the complex structure of contracts. It is worth mentioning that CGE performs well in terms of detection efficiency, thanks to its graph neural network's ability to quickly analyze complex relationships between contracts. Although CGE's overall detection time is slightly longer than our model, its efficiency and accuracy in handling graph data are still commendable.

4.6.6 Method Scalability Experiment

This section aims to experiment and evaluate the scalability of our method in handling large-sample datasets. To achieve this goal, we use the smart contracts in Table 4.3 to construct two specialized training sets: training set α (for reentrancy detection) and training set β (for integer overflow detection). As shown in Table 4.5, training set α contains 150 safe contracts and 200 reentrancy contracts, while training set β contains 150 safe contracts and 200 integer overflow contracts. Test set γ is collected from the SmartBugs [12], containing 200 labeled smart contracts, for evaluating the model's performance.

Computational Efficiency

To evaluate the computational efficiency of our model, we design and conduct two sets of independent training experiments targeting smart contracts with reentrancy and integer overflow vulnerabilities, respectively. Each set of experiments adopts an incremental data experimental design. Each experiment increases 35 smart contracts based on the previous experiment until

Table 4.5: Summary of Datasets in Method Scalability Experiment.

Type of Dataset	**Number of Smart Contracts**			
	Safe	**Reentrancy**	**Integer Overflow**	**Total**
Training Set α	150	200	0	350
Training Set β	150	0	200	350
Test Set γ	120	40	40	200

all 350 contracts are used. Each set of experiments is conducted ten times using the same algorithm and parameter settings.

The experimental results are shown in Figure 4.19, as a scatter plot, where the horizontal axis represents the number of training data, and the vertical axis represents the training time. The figure shows that as the number of training data increases, the training time of both sets of experiments shows an upward trend. By analyzing the trend line in the figure, it is obvious that there is a significant positive correlation between the training time and the number of data. This result is consistent with our expectation because more training data means that the model needs more computational resources to learn and adapt to these data.

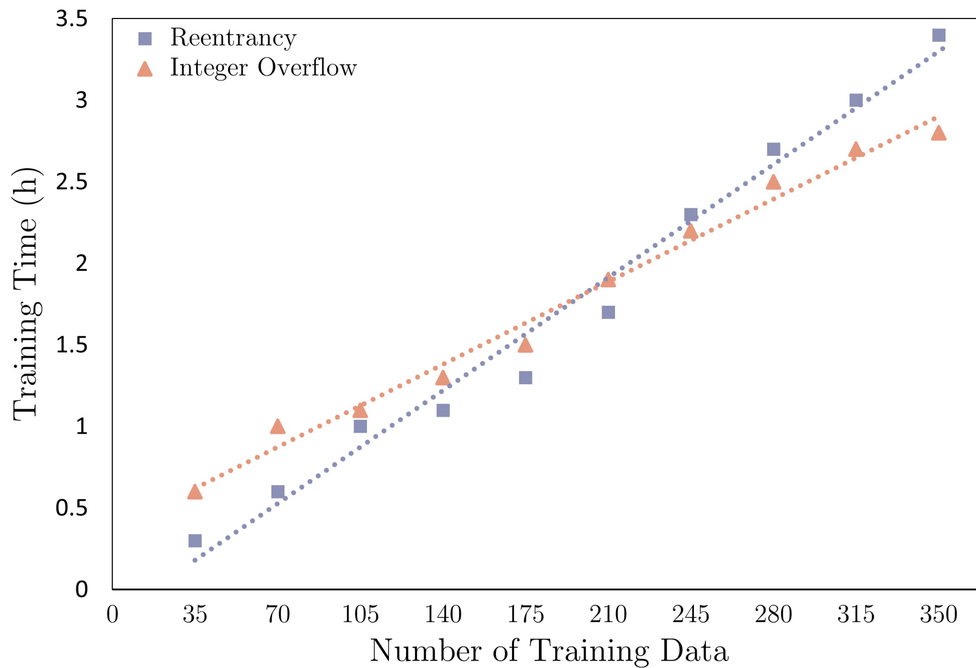


Figure 4.19: Method Computational Efficiency Experiment.

Data Requirement

In the data requirement experiment, we also adopt an incremental data experimental design. The experimental results are shown as a scatter plot, where the horizontal axis is the number of training data, and the vertical axis is the detection accuracy of our model. A smooth curve is used to visualize the correlation between the detection accuracy and the number of training data. As shown in Figure 4.20, the detection accuracy of the model shows a power function-like growth trend as the amount of training data increases.

When the number of data is small, the model's accuracy is rapidly improved, indicating that the model has a good initial learning ability. When the number of data is large, although the accuracy improvement speed slows down, the model can continue to learn and improve its performance, which indicates that the model has the potential to handle more complex datasets. This continuous growth pattern shows that our model can maintain a high detection accuracy when facing the growing data scale, thus proving its scalability in practical applications.

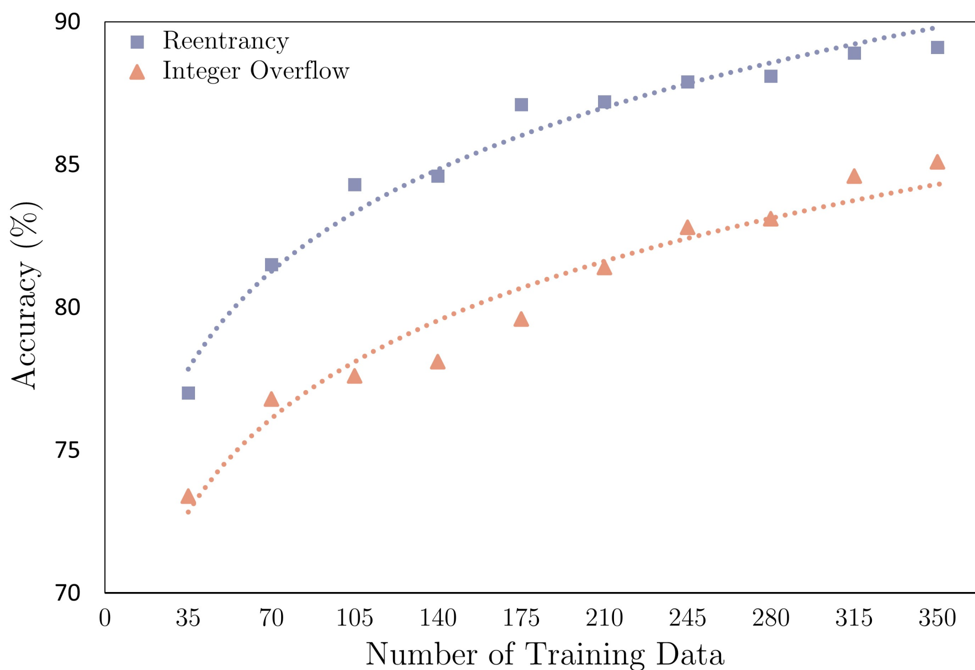


Figure 4.20: Method Data Requirement Experiment.

Testing Efficiency

To validate the testing efficiency of our model, we establish a new comparative experiment. More specifically, the training data from Table 4.5 are utilized to train our models for detecting reentrancy and integer overflow. We conduct incremental testing experiments alongside the four models mentioned in Section 4.6.2. As illustrated in Figure 4.21, the experimental results are presented in a line graph, with the number of testing data on the x-axis and the total detection time on the y-axis. "RE" denotes reentrancy, whereas "IO" signifies integer overflow. These terms refer to specific types of vulnerabilities the model can detect. The results show that as the number of testing data increases, the total detection time for Oyente, Securify, and sFuzz fluctuates and rises due to their detection efficiency being affected by data complexity. In contrast, the GNN-based CGE and our model demonstrate a certain linear relationship between total detection time and the number of testing data. Moreover, our model exhibits

a commendable level of testing efficiency for both reentrancy and integer overflow, capable of handling the challenges of potential large-scale data.

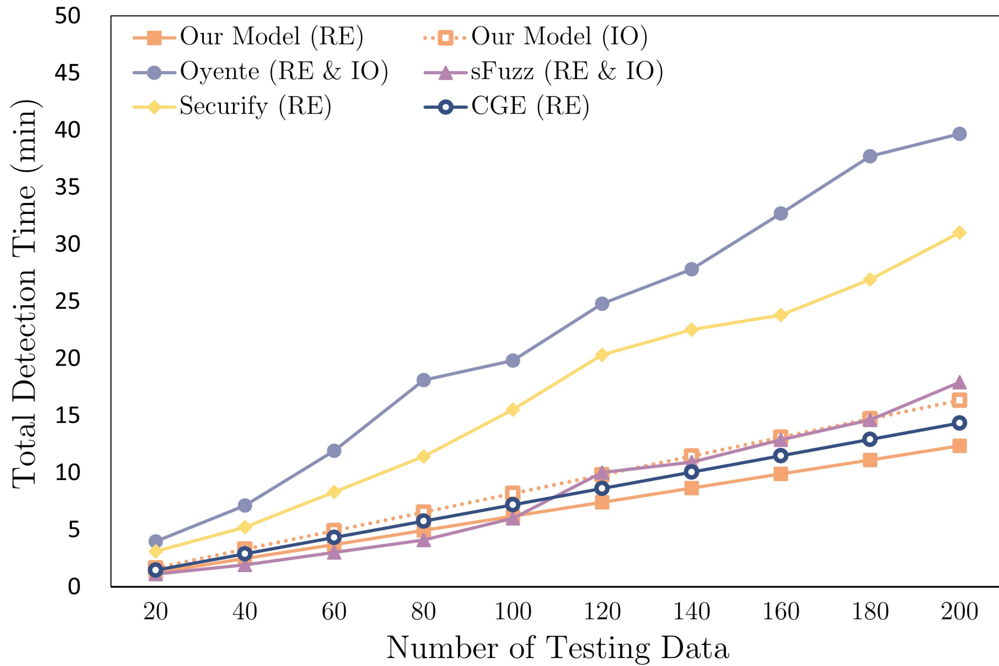


Figure 4.21: Method Testing Efficiency Experiment.

Scalability Discussion

Through experimental analysis, we draw a positive conclusion about the scalability of the proposed method in handling large-sample datasets. The experimental results illustrate that the method has positive adaptability and continuous performance improvement when facing increased data samples.

Regarding computational efficiency, our model has a high training efficiency on small-sample datasets. Besides, on large-sample datasets, although the training time increases, the model still can maintain a relatively stable performance improvement. However, this growth trend also implies the model's sensitivity to the data sample, meaning that the model can handle larger datasets through appropriate resource allocation and optimization strategies while maintaining high accuracy.

In terms of data requirement, the detection accuracy of the model continues to increase as the amount of training data increases, although the growth speed slows down. This phenomenon indicates that the model is constantly learning and absorbing new data features, and even when the data sample is large, the model can still improve performance from the additional data. This continuous but slow accuracy growth reflects the robustness and potential scalability of the model in handling large-sample data.

As to testing efficiency, the model exhibits a linear relationship between the total detection time and the number of testing data, indicating its capability to process increasing data at a consistent rate. The experiment demonstrates that the model can efficiently handle large datasets without significantly reducing performance, a key characteristic of a highly scalable system.

In conclusion, our method shows its potential in handling large-sample smart contract datasets. To further enhance the scalability of the model, future work may consider strategies within distributed computing, such as data parallelism and model parallelism, alongside approaches in federated learning [118]–[120]. These strategies will help improve the model's training efficiency on large-sample datasets without sacrificing accuracy.

4.7 Conclusion

This chapter proposes a smart contract vulnerability detection method based on GAN and GMN for the first time. The method utilizes the structural characteristics of graph data and the cross-graph attention mechanism of the GMN model to identify smart contracts with reentrancy and integer overflow vulnerabilities. By using GraphGAN, we can generate a large number of synthetic CGs, which can maintain similar structural and semantic features to real contracts. The data starvation problem in deep learning is solved by GraphGAN, which makes it possible for GMN to train with small sample data. The detection model achieves high detection accuracy with the cross-graph attention mechanism of GMN. The experimental results demonstrate the proposed smart contract detection method's feasibility, effectiveness, and scalability. This chapter not only presents a novel detection method for the field of smart contract security but also offers new insights into the fusion of GAN and GMN.

Chapter 5

Conclusions and Future Work

Blockchain technology, revolutionizing digital transactions, underpins platforms like Ethereum, one of the most widely used for deploying smart contracts. The growing number of these contracts on Ethereum, carrying increasingly significant financial stakes, underscores the importance of their security. Vulnerabilities in smart contracts, if exploited, can severely damage Ethereum's trustworthiness and cause substantial economic losses to investors. The unalterable nature of smart contracts necessitates rigorous and proactive vulnerability checks to preempt potential vulnerabilities. In recent years, as smart contract technology has rapidly advanced, the security of these digital contracts has garnered escalating attention, highlighting an urgent need for effective detection and prevention strategies.

This thesis investigates current smart contract vulnerability detection methods and identifies key limitations in traditional approaches, such as reliance on expert rules, low automation, and lengthy detection times. While emerging Machine Learning-based techniques for smart contract vulnerability detection have been increasingly adopted due to their efficiency in processing large-scale data, they face two primary challenges: transforming smart contracts into machine learning-compatible feature inputs and enhancing vulnerability detection accuracy. Moreover, obtaining smart contract codes for machine learning presents several challenges. These include the limited availability of open-source contracts and privacy and legal issues. Additionally, the process is costly in terms of time, resources, and the effort required for manual filtering. These issues often result in insufficient experimental data, adversely affecting the accuracy and generalizability of detection models.

To address the limitations in existing methods for detecting vulnerabilities in smart contracts, this thesis explores the following research topics:

1) Developing a machine learning-oriented code representation scheme for smart contract source codes, aiming to retain crucial information and vulnerability features as much as possible.

2) Investigating and implementing a few-shot learning-based vulnerability detection approach to overcome the challenge of data starvation in machine learning.

3) Addressing the limitations of traditional vulnerability detection methods by researching and implementing an efficient solution based on graph neural networks.

5.1 Summary of Outcomes

In response to the current state of smart contract security and the research motivation mentioned earlier, this thesis has completed research work in the following three aspects:

1) This thesis conducts an in-depth study of the smart contract vulnerability detection domain, including fundamental concepts of smart contracts, security status, and the classification and characteristics of ten common vulnerabilities in three levels. Additionally, the thesis surveys mainstream smart contract vulnerability detection methods, analyzing the shortcomings of the current three types of traditional methods from a technical principle perspective, and introduces and analyzes the research value of the methods based on machine learning.
2) The thesis proposes a novel integer overflow vulnerability detection method based on code embedding and Generative Adversarial Networks (GAN). This approach integrates vector representations of code with GAN discriminator feedback to enhance the accuracy and efficiency of detecting integer overflow vulnerabilities in smart contracts. By using GAN to augment a small-sample training set, the model achieves efficient vulnerability detection based on few-shot learning. This research offers new insights into code representation and few-shot learning in the field of smart contract vulnerability detection.
3) The thesis introduces a smart contract vulnerability detection method based on the GAN and Graph Matching Networks (GMN), effectively identifying reentrancy and integer overflow vulnerabilities in Ethereum smart contracts. By converting the Solidity code of smart contracts into Contract Graphs (CG) containing semantic and structural information, we utilize a GAN model based on graph representation to generate a large number of synthetic contract graphs, providing technical support for few-shot learning in subsequent GMN training. Subsequently, these synthetic graphs are used to train the GMN model, which employs a cross-graph attention mechanism to calculate feature similarity between target contracts and vulnerability contracts. Experimental results demonstrate that this approach surpasses traditional detection methods in identifying reentrancy and integer overflow vulnerabilities, validating the significant role of graph representation, few-shot learning, and graph neural networks in smart contract vulnerability detection.

5.2 Recommendations & Future Work

The smart contract vulnerability detection methods proposed in this thesis achieve our research aims, but there are still two potential directions for innovation.

Multi-Layer Vulnerability Detection Using GNNs

This research preliminarily explores the potential of Graph Neural Networks (GNN) in smart contract source code vulnerability detection. However, many details and problems still need further research and optimization. For example, how to better construct CGs to reflect the characteristics and vulnerabilities of smart contracts; how to choose suitable GNN models to adapt to different layers of vulnerability detection tasks; how to combine expert knowledge to enhance the accuracy and interpretability of GNN models; how to evaluate and compare the effectiveness and efficiency of different GNN models in smart contract vulnerability detection. Future work will conduct vulnerability detection and analysis from three perspectives: source code layer, EVM execution layer, and blockchain system layer, using different GNN models to

fully mine the source code, bytecode, and on-chain interaction information of smart contracts, thereby improving the accuracy, coverage, efficiency, and automation of vulnerability detection. Specifically, future work will include the following aspects:

- Research on converting smart contract source code to graph, where nodes represent key functions and variables, and edges represent control flow, data flow, and program flow relationships. Optimize the node classification model based on GMN to identify vulnerable nodes in the graph better. In addition to integer overflow and reentrancy, construct feature rules for other common types of vulnerabilities. Integrate expert knowledge by combining rules and patterns related to vulnerabilities to improve the accuracy and interpretability of the model.
- Convert EVM bytecode to graph, where nodes represent instructions and edges represent execution order. Construct a graph classification model based on the Graph Convolutional Network (GCN) [116] to detect vulnerabilities in the bytecode graph. GCN is a GNN model that can learn node and global features on the graph, and update node representations by aggregating neighbor information. Future work will use the GCN to learn the features of the bytecode graph and judge whether the graph contains vulnerabilities.
- With the emergence of more and more blockchain platforms and protocols, interoperability has become an important challenge and opportunity. Interoperability refers to the ability of different blockchain systems to communicate and exchange information with each other, which is essential for achieving cross-chain value transfer and data sharing. In this development context, smart contract vulnerability detection may need to consider the information interaction problem between different systems. More specifically, we can try to analyze the interaction graph formed by the deployment and interaction of smart contracts on the blockchain, where nodes represent contract instances, and edges represent contract calls. Develop a link prediction model based on the Graph Attention Network (GAT) [121]. GAT is a GNN model that can learn the importance of nodes and edges on the graph, and it can weight neighbor information by assigning different attention weights. Future work will use the GAT to learn the structure of the interaction graph and predict whether there are potential vulnerability links in the interaction graph.

Few-Shot Learning in Smart Contract Vulnerability Detection

This research preliminarily explores the data augmentation capability of GAN to assist GMN in smart contract vulnerability detection. Facing data ethics risks and data labeling difficulties, how to use a small amount of labeled data and a large amount of unlabeled data to improve the generalization and robustness of GNN models is still a very meaningful and challenging research problem. Future work will explore different methods to achieve the goal of few-shot learning in smart contract vulnerability detection. To be specific, future work will include the following aspects:

- Use the adversarial learning method of the GAN to train the GCN model. GraphGAN algorithm [104] will be used to maximize the similarity within the same class of samples and minimize the similarity between different classes. Future work will involve using GraphGAN to generate new bytecode graphs, which will increase the diversity of training data. Additionally, we will combine this approach with the GCN to implement a few-shot learning vulnerability detection method.
- Apply the Model Agnostic Meta-Learning (MAML) [122] to the GMN model to achieve

fast adaptation to new tasks. MAML is an algorithm that can perform meta-learning on multiple tasks, and it can optimize model parameters by performing gradient updates on multiple tasks, so that the model can adapt to new tasks in a few steps. Future work will use different types of smart contract vulnerabilities as different tasks, thereby enhancing the model's generalization ability in source code layer vulnerability detection.

These two research directions are not only expected to consolidate the development98 prospects of blockchain technology, but also provide a roadmap for GNN and few-shot learning to advance the field of smart contract security. By focusing on these areas, future work is expected to develop more powerful and efficient vulnerability detection methods to ensure blockchain technology's continuous development and trust.

Chapter 6

Publication

1. **Hao Li**, Xu Wang, Guangsheng Yu, Wei Ni, and Ren Ping Liu, "A Generative Adversarial Networks-Based Integer Overflow Detection Model for Smart Contracts," in 2023 22nd International Symposium on Communications and Information Technologies (ISCIT), IEEE, 2023, pp. 31-36.

2. **Hao Li**, Xu Wang, Guangsheng Yu, Wei Ni, Ren Ping Liu, Nektarios Georgalas, and Andrew Reeves, "Smart Contract Vulnerability Detection Based on Generative Adversarial Networks and Graph Matching Networks," in 2023 2nd International Conference on Network Simulation and Evaluation (NSE), Springer, 2023.

Bibliography

[1] N. Szabo, "Formalizing and securing relationships on public networks," *First monday*, 1997.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized business review*, 2008.

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

[4] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *Financial Cryptography and Data Security: FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers 20*, Springer, 2016, pp. 79–94.

[5] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," *Future generation computer systems*, vol. 107, pp. 841–853, 2020.

[6] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons, "Smart contracts vulnerabilities: A call for blockchain software engineering?" In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, IEEE, 2018, pp. 19–25.

[7] Y. Tsuchiya and N. Hiramoto, "How cryptocurrency is laundered: Case study of coincheck hacking incident," *Forensic Science International: Reports*, vol. 4, p. 100 241, 2021.

[8] V. Kustov, G. Aleksey, B. Nikolay, S. Ekaterina, and R. V. Ravi, "Three sources of blockchain technology vulnerabilities-how to deal with them?" In *2022 Second International Conference on Computer Science, Engineering and Applications (ICCSEA)*, IEEE, 2022, pp. 1–8.

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

[10] 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.

[11] 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.

[12] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, 2020, pp. 530–541.

[13] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *2017 3rd IEEE international conference on computer and communications (ICCC)*, IEEE, 2017, pp. 1298–1302.

[14] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, "A comparative study of deep learning-based vulnerability detection system," *IEEE Access*, vol. 7, pp. 103 184–103 197, 2019.

[15] Y. Yu, X. Si, C. Hu, and J. Zhang, "A review of recurrent neural networks: Lstm cells and network architectures," *Neural computation*, vol. 31, no. 7, pp. 1235–1270, 2019.

[16] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, vol. 1, 2015, pp. 858–868.

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

[18] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, "Combining graph neural networks with expert knowledge for smart contract vulnerability detection," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[19] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *International conference on machine learning*, PMLR, 2019, pp. 3835–3845.

[20] 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.

[21] Z. Zhang, X. Wang, G. Yu, *et al.*, "A community detection-based blockchain sharding scheme," in *International Conference on Blockchain*, Springer, 2022, pp. 78–91.

[22] I. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[23] 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.

[24] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, 2016.

[25] H. Wang, J. Wang, J. Wang, *et al.*, "Learning graph representation with generative adversarial nets," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 8, pp. 3090–3103, 2019.

[26] X. Wang, G. Yu, R. P. Liu, *et al.*, "Blockchain-enabled fish provenance and quality tracking system," *IEEE Internet of Things Journal*, vol. 9, no. 11, pp. 8130–8142, 2021.

[27] W. Yang, S. Wang, X. Yin, X. Wang, and J. Hu, "A review on security issues and solutions of the internet of drones," *IEEE Open Journal of the Computer Society*, vol. 3, pp. 96–110, 2022.

[28] B. Ma, X. Lin, X. Wang, *et al.*, "New cloaking region obfuscation for road network-indistinguishability and location privacy," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, 2022, pp. 160–170.

[29] 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.

[30] C. Metz, *The biggest crowdfunding project ever—the dao—is kind of a mess*, 2016.
[31] C. Wu, J. Xiong, H. Xiong, Y. Zhao, and W. Yi, "A review on recent progress of smart contract in blockchain," *IEEE Access*, vol. 10, pp. 50 839–50 863, 2022.
[32] V. D'silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.
[33] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.
[34] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *2010 IEEE Symposium on Security and Privacy*, IEEE, 2010, pp. 497–512.
[35] P. Godefroid, P. de Halleux, A. V. Nori, *et al.*, "Automating software testing using program analysis," *IEEE software*, vol. 25, no. 5, pp. 30–37, 2008.
[36] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software," in *NDSS*, Citeseer, vol. 5, 2005, pp. 3–4.
[37] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.
[38] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.
[39] X. Wang, X. Zha, W. Ni, *et al.*, "Survey on blockchain for internet of things," *Computer Communications*, vol. 136, pp. 10–29, 2019.
[40] J. Wang, T. Zhang, N. Sebe, H. T. Shen, *et al.*, "A survey on learning to hash," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 4, pp. 769–790, 2017.
[41] Y. Yuan and F.-Y. Wang, "Blockchain and cryptocurrencies: Model, techniques, and applications," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 48, no. 9, pp. 1421–1428, 2018.
[42] X. Wang, W. Ni, X. Zha, *et al.*, "Capacity analysis of public blockchain," *Computer Communications*, vol. 177, pp. 112–124, 2021.
[43] *Ethereum - a global, decentralized platform for money and new kinds of applications*, `https://https://ethereum.org`.
[44] *Hyperledger - the open source, global ecosystem for enterprise-grade blockchain technologies*, `https://www.hyperledger.org`.
[45] *Eosio - fast, flexible, and forward-driven*, `https://eos.io`.
[46] *Cardano - a proof-of-stake blockchain platform*, `https://cardano.org`.
[47] *Tron - decentralize the web*, `https://tron.network`.
[48] *Stellar - a blockchain network for payments and tokenization*, `https://stellar.org`.
[49] *Neo - resilience, meet innovation*, `https://neo.org`.
[50] *Tezos - a blockchain designed to evolve*, `https://tezos.com`.
[51] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 3–16.
[52] Ethereum, *The merge*, `https://ethereum.org/roadmap/merge`.
[53] *Vyper - a contract-oriented, pythonic programming language that targets the ethereum virtual machine (evm)*, `https://docs.vyperlang.org/en/stable`.

[54] *Yul - solidity 0.8.25 documentation*, https://docs.soliditylang.org/en/latest/yul.html.
[55] R. Böhme, N. Christin, B. Edelman, and T. Moore, "Bitcoin: Economics, technology, and governance," *Journal of economic Perspectives*, vol. 29, no. 2, pp. 213–238, 2015.
[56] *Uniswap protocol - swap, earn, and build on the leading decentralized crypto trading protocol*, https://uniswap.org.
[57] *Markerdao - an unbiased global financial system*, https://makerdao.com.
[58] *Augur - your global, no-limit betting platform*, https://augur.net.
[59] *Uport - open identity system for the decentralized web*, https://www.uport.me.
[60] *Medrec - your personal electronic health record*, https://medrec-m.com.
[61] X. Wang, G. Yu, X. Zha, *et al.*, "Capacity of blockchain based internet-of-things: Testbed and analysis," *Internet of Things*, vol. 8, p. 100 109, 2019.
[62] *Iota - an open, feeless data and value transfer protocol*, https://www.iota.org.
[63] *Solidity - a statically-typed curly-braces programming language designed for developing smart contracts that run on ethereum*, https://soliditylang.org.
[64] *Flint - a new type-safe, contract-oriented programming language specifically designed for writing robust smart contracts on ethereum*, https://github.com/flintlang/flint.
[65] *Obsidian - an object-oriented, strongly-typed language for smart contracts*, https://github.com/mcoblenz/Obsidian.
[66] Z. Yang and H. Lei, "Lolisa: Formal syntax and semantics for a subset of the solidity programming language in mathematical tool coq," *Mathematical Problems in Engineering*, vol. 2020, pp. 1–15, 2020.
[67] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6*, Springer, 2017, pp. 164–186.
[68] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Systematic review of security vulnerabilities in ethereum blockchain smart contract," *IEEE Access*, vol. 10, pp. 6605–6621, 2022.
[69] D. He, R. Wu, X. Li, S. Chan, and M. Guizani, "Detection of vulnerabilities of blockchain smart contracts," *IEEE Internet of Things Journal*, 2023.
[70] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart contract: Attacks and protections," *IEEE Access*, vol. 8, pp. 24 416–24 427, 2020.
[71] *Mythril - a security analysis tool for evm bytecode*, https://github.com/ConseSys/mythril.
[72] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural networks," in *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, 2021, pp. 3283–3290.
[73] M. Zalewski, *American fuzzy lop - a security-oriented fuzzer*, https://github.com/google/AFL, 2020.
[74] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 259–269.
[75] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 531–548.

[76] Z. Liu, P. Qian, J. Yang, *et al.*, "Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 1237–1251, 2023.
[77] H. Yang, X. Gu, X. Chen, L. Zheng, and Z. Cui, "Crossfuzz: Cross-contract fuzzing for smart contract vulnerability detection," *Science of Computer Programming*, vol. 234, p. 103 076, 2024.
[78] C. Shou, S. Tan, and K. Sen, "Ityfuzz: Snapshot-based fuzzer for smart contract," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 322–333.
[79] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th annual computer security applications conference*, 2018, pp. 653–663.
[80] P. Zheng, Z. Zheng, and X. Luo, "Park: Accelerating smart contract vulnerability detection via parallel-fork symbolic execution," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 740–751.
[81] A. Ghaleb, J. Rubin, and K. Pattabiraman, "Achecker: Statically detecting smart contract access control vulnerabilities," *Proc. ACM ICSE*, 2023.
[82] M. Pasqua, A. Benini, F. Contro, M. Crosara, M. Dalla Preda, and M. Ceccato, "Enhancing ethereum smart-contracts static analysis by computing a precise control-flow graph of ethereum bytecode," *Journal of Systems and Software*, vol. 200, p. 111 653, 2023.
[83] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, *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.
[84] E. Hildenbrandt, M. Saxena, N. Rodrigues, *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, IEEE, 2018, pp. 204–217.
[85] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
[86] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts.," in *Ndss*, 2018, pp. 1–12.
[87] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework," in *International Conference on Computer Aided Verification*, Springer, 2015, pp. 343–361.
[88] M. Almakhour, L. Sliman, A. E. Samhat, and A. Mellouk, "A formal verification approach for composite smart contracts security using fsm," *Journal of King Saud University - Computer and Information Sciences*, vol. 35, no. 1, pp. 70–86, 2023.
[89] H. Deng, X. Wang, G. Yu, X. Dang, and R. P. Liu, "A novel weights-less watermark embedding method for neural network models," in *2023 22nd International Symposium on Communications and Information Technologies (ISCIT)*, 2023, pp. 25–30. DOI: 10.1109/ISCIT57293.2023.10376108.
[90] J. Zhou, G. Cui, S. Hu, *et al.*, "Graph neural networks: A review of methods and applications," *AI open*, vol. 1, pp. 57–81, 2020.
[91] T. Altaf, X. Wang, W. Ni, G. Yu, R. P. Liu, and R. Braun, "A new concatenated multigraph neural network for iot intrusion detection," *Internet of Things*, vol. 22, p. 100 818, 2023.

[92] G. Yu, Q. Wang, T. Altaf, X. Wang, X. Xu, and S. Chen, "Predicting nft classification with gnn: A recommender system for web3 assets," in *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, IEEE, 2023, pp. 1–5.

[93] J. Cai, B. Li, J. Zhang, X. Sun, and B. Chen, "Combine sliced joint graph with graph neural networks for smart contract vulnerability detection," *Journal of Systems and Software*, vol. 195, p. 111 550, 2023.

[94] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: Analysis, applications, and prospects," *IEEE transactions on neural networks and learning systems*, 2021.

[95] S.-J. Hwang, S.-H. Choi, J. Shin, and Y.-H. Choi, "Codenet: Code-targeted convolutional neural network architecture for smart contract vulnerability detection," *IEEE Access*, vol. 10, pp. 32 595–32 607, 2022. DOI: 10.1109/ACCESS.2022.3162065.

[96] G. Van Houdt, C. Mosquera, and G. Nápoles, "A review on the long short-term memory model," *Artificial Intelligence Review*, vol. 53, pp. 5929–5955, 2020.

[97] R. Dey and F. M. Salem, "Gate-variants of gated recurrent unit (gru) neural networks," in *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)*, IEEE, 2017, pp. 1597–1600.

[98] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang, "Towards automated reentrancy detection for smart contracts based on sequential models," *IEEE Access*, vol. 8, pp. 19 685–19 695, 2020.

[99] Y. Sun, D. Wu, Y. Xue, *et al.*, "When gpt meets program analysis: Towards intelligent detection of smart contract logic vulnerabilities in gptscan," *arXiv preprint arXiv:2308.03 314*, 2023.

[100] H. Wu, H. Dong, Y. He, and Q. Duan, "Smart contract vulnerability detection based on hybrid attention mechanism model," *Applied Sciences*, vol. 13, no. 2, p. 770, 2023.

[101] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 783–794.

[102] F. Bond, *Solidity-parser-antlr: A solidity parser built on top of a robust antlr4 grammar*, https://github.com/federicobond/solidity-parser-antlr, 2019.

[103] X. Rong, "Word2vec parameter learning explained," *arXiv preprint arXiv:1411.2738*, 2014.

[104] H. Wang, J. Wang, J. Wang, *et al.*, "Graphgan: Graph representation learning with generative adversarial nets," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, 2018.

[105] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels.," *Journal of Machine Learning Research*, vol. 12, no. 9, 2011.

[106] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities: A survey," *arXiv preprint arXiv:1908.08605*, 2019.

[107] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2874–2891, 2020.

[108] G. Yu, X. Wang, P. Yu, C. Sun, W. Ni, and R. P. Liu, "Dataset obfuscation: Its applications to and impacts on edge machine learning," *arXiv preprint arXiv:2208.03909*, 2022.

[109] G. Yu, X. Wang, C. Sun, P. Yu, W. Ni, and R. P. Liu, "Obfuscating the dataset: Impacts and applications," *ACM Transactions on Intelligent Systems and Technology*, vol. 14, no. 5, pp. 1–15, 2023.

[110] 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.

[111] 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.

[112] F. Bond, *Etherscan—the ethereum blockchain explorer*, `https://etherscan.io`, 2019.

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

[114] I. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, "Generative adversarial networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[115] F. Bond, *Solidity-parser*, `https://github.com/solidity-parser/parser`.

[116] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[117] T. Altaf, X. Wang, W. Ni, R. P. Liu, and R. Braun, "Ne-gconv: A lightweight node edge graph convolutional network for intrusion detection," *Computers & Security*, vol. 130, p. 103 285, 2023.

[118] Y. Jiang, B. Ma, X. Wang, *et al.*, "A secure aggregation for federated learning on long-tailed data," *arXiv preprint arXiv:2307.08324*, 2023.

[119] G. Yu, X. Wang, C. Sun, *et al.*, "Ironforge: An open, secure, fair, decentralized federated learning," *IEEE Transactions on Neural Networks and Learning Systems*, 2023.

[120] Y. Jiang, B. Ma, X. Wang, *et al.*, "Blockchained federated learning for internet of things: A comprehensive survey," *arXiv preprint arXiv:2305.04513*, 2023.

[121] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, Y. Bengio, *et al.*, "Graph attention networks," *stat*, vol. 1050, no. 20, pp. 10–48 550, 2017.

[122] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *International conference on machine learning*, PMLR, 2017, pp. 1126–1135.