Graph Representation Learning for Security Analytics in Decentralized Software Systems and Social Networks

Von der Fakultät für Elektrotechnik und Informatik
der Gottfried Wilhelm Leibniz Universität Hannover
zur Erlangung des akademischen Grades

DOKTOR DER NATURWISSENSCHAFTEN

Dr. rer. nat.

vorgelegte Dissertation

von

M. Eng. Huu Hoang Nguyen

geboren am 21. Januar 1990, in Dong Thap, Vietnam

2024

Referent: Prof. Dr. -Ing Wolfgang Nejdl
Korreferent: Prof. Dr. Jiang Lingxiao
Vorsitz: Prof. Dr. Ziawasch Abedjan
Tag der Promotion: 03.05.2024

"The steps you take don't need to be big; they just need to take you in the right direction."

Jemma Simmons (Marvel Cinematic Universe)

ABSTRACT

With the rapid advancement in digital transformation, various daily interactions, transactions, and operations typically depend on extensive network-structured systems. The inherent complexity of these platforms has become a critical challenge in ensuring their security and robustness, with impacts spanning individual users to large-scale organizations. Graph representation learning has emerged as a potential methodology to address various security analytics within these complex systems, especially in software code and social network analysis, and its applications in criminology. For software code, graph representations can capture the information of control-flow graphs and call graphs, which can be leveraged to detect vulnerabilities and improve software reliability. In the case of social network analysis in criminal investigation, graph representations can capture the social connections and interactions between individuals, which can be used to identify key players, detect illegal activities, and predict new/unobserved criminal cases.

In this thesis, we focus on two critical security topics using graph learning-based approaches: (1) addressing criminal investigation issues and (2) detecting vulnerabilities of Ethereum blockchain smart contracts. First, we propose the SoChainDB database, which facilitates obtaining data from blockchain-based social networks and conducting extensive analyses to understand Hive blockchain social data. Moreover, to apply social network analysis in criminal investigation, two graph-based machine learning frameworks are presented to address investigation issues in a burglary use case, one being transductive link prediction and the other being inductive link prediction. Then, we propose MANDO, an approach that utilizes a new heterogeneous graph representation of control-flow graphs and call graphs to learn the structures of heterogeneous contract graphs. Building upon MANDO, two deep graph learning-based frameworks, MANDO-GURU and MANDO-HGT, are proposed for accurate vulnerability detection at both the coarse-grained contract and fine-grained line levels. Empirical results show that MANDO frameworks significantly improve the detection accuracy of other state-of-the-art techniques for various vulnerability types in either source code or bytecode.

Keywords: *graph embedding, graph neural network, heterogeneous graph learning, decentralized social network, vulnerability detection, blockchain, smart contract, social network analysis, crime linkage, link prediction, database*

ZUSAMMENFASSUNG

Mit dem rasanten Fortschritt der digitalen Transformation hängen verschiedene tägliche Interaktionen, Transaktionen und Abläufe in der Regel von umfangreichen, netzwerkstrukturierten Systemen ab. Die inhärente Komplexität dieser Plattformen stellt eine kritische Herausforderung dar, um ihre Sicherheit und Stabilität zu gewährleisten, mit Auswirkungen, die von Einzelpersonen bis hin zu großflächigen Organisationen reichen. Graph-Representation-Learning hat sich als eine vielversprechende Methodik etabliert, um vielfältige Sicherheitsanalysen in diesen komplexen Systemen anzugehen, insbesondere in der Softwarecodeanalyse und der sozialen Netzwerkanalyse, sowie in deren Anwendungen in der Kriminologie. Bei Softwarecodes können Graph-Representationen die Informationen von Kontrollflussdiagrammen und Funktionsaufrufgraphen erfassen, die genutzt werden können, um Schwachstellen aufzudecken und die Softwarezuverlässigkeit zu verbessern. Im Bereich der sozialen Netzwerkanalyse in der kriminalpolizeilichen Ermittlung können Graph-Representationen die sozialen Verbindungen und Interaktionen zwischen Individuen abbilden, was dazu verwendet werden kann, Schlüsselpersonen zu identifizieren, illegale Aktivitäten zu erkennen und neue/unbeobachtete kriminelle Fälle vorherzusagen.

In dieser Dissertation konzentrieren wir uns auf zwei kritische Sicherheitsthemen unter Verwendung von Graphen-Learning-basierten Ansätzen: (1) die Behandlung von Problemen kriminaltechnischer Untersuchungen und (2) die Erkennung von Schwachstellen in Ethereum-Blockchain-Smart-Contracts. Zunächst schlagen wir die SoChainDB-Datenbank vor, die das Abrufen von Daten aus Blockchain-basierten sozialen Netzwerken erleichtert und umfangreiche Analysen zur Verständigung der Hive-Blockchain-Sozialdaten ermöglicht. Darüber hinaus, um soziale Netzwerkanalyse in kriminaltechnischen Untersuchungen anzuwenden, werden zwei graphenbasierte maschinelle Lernframeworks vorgestellt, um Untersuchungsprobleme in einem Einbruchsfall zu behandeln, einer davon ist transduktive Link-Prädiktion und der andere induktive Link-Prädiktion. Anschließend schlagen wir MANDO vor, einen Ansatz, der eine neue heterogene Graphen-Representation von Kontrollflussgraphen und Aufrufgraphen nutzt, um die Strukturen heterogener Vertragsgraphen zu erlernen. Aufbauend auf MANDO werden zwei Deep-Graphen-Learningbasierte Frameworks, MANDO-GURU und MANDO-HGT, für eine genaue Schwachstellenerkennung sowohl auf der grobkörnigen Vertrags- als auch auf der feinkörnigen Zeilenebene vorgeschlagen. Empirische Ergebnisse zeigen, dass die MANDO-Frameworks die Erkennungsgenauigkeit anderer State-of-the-Art-Techniken für verschiedene Schwachstellentypen in entweder Source-Code oder Bytecode signifikant verbessern.

Schlüsselwörter: *Grapheneinbettung, Graph-Neuronales-Netzwerk, heterogenes Graphenlernen, dezentrales soziales Netzwerk, Schwachstellenerkennung, Blockchain, Smart Contract, soziale Netzwerkanalyse, Kriminalitätsverknüpfung, Link-Prädiktion, Datenbank*

Acknowledgements

During my doctoral studies over the last three years, I have fortunately received invaluable assistance and contributions from my supervisors, colleagues, friends, and family members. In moments of most challenge and difficulty, their sincere and practical advice has been vital in facilitating the successful progression of my research.

I wish to express my profound gratitude to Prof. Dr. Wolfgang Nejdl for supporting me throughout my doctoral journey and giving me the exciting opportunity to work in an active and excellent research environment at the L3S Research Center. My wholehearted thanks go to Prof. Dr. Jiang Lingxiao from Singapore Management University. His invaluable guidance has steered my ideas and encouraged me to delve deeper into my research endeavors.

I am very grateful to my mentors at L3S Research Center, Dr. Zahra Ahmadi, Dr. Daniel Kudenko, Dr. Marco Fisichella, and Dr. Tuan-Anh Hoang, for constantly supporting and guiding me, especially at the initial stages of my doctoral studies. A very special thanks to my fellows and friends, Dr. Thanh-Nam Doan and Nhat-Minh Nguyen, for our collaborative efforts culminating in outstanding scientific achievements.

Lastly, my deepest thanks are reserved for my wife, Hong Lan Thao Le, whose unwavering support and provision of the best conditions for me to maintain my focus on research during the last years.

Contents

List of Figures

List of Tables

List of Abbreviations

ML	Machine Learning
DL	Deep Learning
LSTM	Long Short Term Memory
GNN	Graph Neural Network
MLP	Multi-Layer Perceptron
CFG	Control-Flow Graph
HCFG	Heterogeneous Control-Flow Graph
CG	Call Graph
HCG	Heterogeneous Call Graph
ACC	Accuracy
AP	Average Precision
ROC	Receiver Operating Characteristic
AUC	Area Under The Curve

Dedicated to my family and my curiosity

Chapter 1

General Introduction

1.1 Overview of Graph Representation Learning

Graph representation learning, a specialized subfield of machine learning, is increasingly proving effective in modeling and analyzing complex graph data structures [1]. Like other machine learning models, it allows learning features from the data to generate predictive models suitable for specified prediction tasks. However, graph representation learning has an advantage over other machine learning models through its ability to represent and learn useful features on complex data associations, especially in graph-structured data. With the popularity of graph data in many fields in recent years, such as social networks, criminal networks, protein interactions in biology, or function invocations in software engineering, this research direction is opening up several new approaches for practical downstream tasks such as link prediction, community detection, social influence analysis, data or entity classifications [2]. The complexity of the structured data derives from the relational nature of entities and their linkages. For instance, this complexity manifests in criminal networks through offenders with various relationships, including their friends, neighborhoods, family ties, or gang affiliations. Similarly, decentralized software systems, e.g., blockchain networks, often include multiple objects such as users, public accounts, or smart contracts, and their connections represent complex relationships between them, which could be cryptocurrency transfer flows of the users or function invocation associations inside a smart contract.

Machine learning models mainly deal with many data types, including structured data (e.g., tables) or unstructured data (e.g., text or images). The general machine learning algorithms assume the data points are independently and identically distributed [3]. This assumption might suit many traditional data types, but it becomes less applicable when handling complex structured data as graphs, where data points (i.e., nodes) are associated. In contrast, the data instances from a network or a graph are not independent but interconnected. Thus, graph representation learning methods are proposed to encode graphs into low-dimensional vector spaces, which preserve structural and attribute information. The main focus of graph representation learning is to learn a function that possibly maps nodes, edges, or entire graphs to vector representations, called node embeddings or graph embeddings, depending on the targeted downstream tasks. Accordingly, these approaches provide a way to convert the rich but complex information contained in graphs into a format that can be easily used by traditional machine learning models [4].

Graph representation learning techniques span a wide array of methods, each with distinct mechanisms and goals. Although the categorization into unsupervised and supervised techniques is often cited, it is crucial to recognize that the boundary between these categories is somewhat blurred in practice [2], with many techniques sharing elements of both.

Fundamental graph embedding methods aiming primarily to learn representations that capture the structural properties of the graph draw parallels to unsupervised learning techniques in traditional machine learning. Techniques such as DeepWalk [5], node2vec [6], and LINE [7] utilize processes like

random walks, matrix factorization or graph-based regularization to derive node embeddings. The main focus of these methods lies in preserving the topological properties of the graph, and they operate independently of specific labels or attributes for training.

On the other hand, some newer methods employ graph neural networks to learn node embeddings in an end-to-end manner. Besides capturing the graph's topologies, these methods leverage specific attributes or labels associated with the graph nodes or edges. Approaches including Graph Convolutional Networks (GCNs) [8], Graph Attention Networks (GATs) [9], and GraphSAGE [10] bear similarities to (semi-)supervised learning in traditional machine learning. They concentrate on learning representations being beneficial for predicting specific attributes or labels.

Nevertheless, some graph representation learning techniques integrate elements of both structural and attribute-based learning, placing themselves somewhere between these two kinds. These combined methods aim to make the most of both strategies, effectively capturing both the structural properties of the graph and the specific attributes or labels associated with the nodes or edges. The selection of the appropriate method largely depends on the specific requirements and constraints of the downstream tasks.

1.2 General Motivation

With the growing availability of graph-structured data and advancements in deep learning techniques, graph representation learning has demonstrated promising results across various domains. This thesis will concentrate on applying graph representation learning for security analytics in social networks and decentralized software systems with a focus on blockchain systems. Specifically, we emphasize criminal investigations and smart contract security analysis.

Decentralized software systems, such as blockchain networks, have gained significant traction due to their potential for enabling secure, transparent, and tamper-resistant data management. However, these systems also present unique security challenges, such as detecting software vulnerabilities in smart contracts or identifying malicious nodes in the network. Graph representation learning can play a crucial role in addressing these challenges by modeling the complex relationships among entities in the system and learning informative representations that can be used for detecting potential security risks.

An emerging area of interest is decentralized social networks built on blockchain technology, which are growing in popularity as they address critical concerns associated with traditional centralized social networks, such as content ownership and over-commercialization. Despite their potential, the rich and valuable data these networks generate remain relatively untapped due to challenges in data collection, which often require specialized blockchain knowledge. This thesis will explore the application of a database framework to facilitate data collection from blockchain-based social networks, further enhancing the efficacy of graph representation learning in these scenarios.

In the field of social network analysis, graph representation learning can be applied to uncover hidden relationships, identify potential illegal activities, and track suspicious entities, ultimately aiding in the prevention, detection, and investigation of criminal activities. Criminals increasingly leverage social networks for various unlawful activities, such as money laundering, terrorist financing, drug trafficking, and human trafficking. By applying graph representation learning techniques to model the complex relationships within social networks, it becomes possible to develop more effective methods for addressing security challenges in these digital platforms.

This thesis aims to contribute to the growing area of research in graph representation learning by focusing on its applications in security analytics for decentralized software systems and social networks. We will investigate various techniques and approaches, ranging from traditional graph embedding methods to more recent deep graph learning-based methods, such as Graph Convolutional Networks (GCNs) [8], Graph Attention Networks (GATs) [9], Heterogeneous Graph Attention Networks (HANs) [11], and Heterogeneous Graph Transformers (HGTs) [12]. By exploring these techniques and their potential applications, this thesis seeks to advance the state-of-the-art in graph representation learning and showcase its potential for addressing critical security challenges in both decentralized software systems and social networks.

1.3 Research Objectives

A key aspect of this thesis will be the development of novel graph representation learning techniques tailored to the specific requirements and challenges of the target domains. In the context of decentralized software systems, we will investigate methods for fine-grained detection of smart contract vulnerabilities and explore the potential of heterogeneous graph transformers for vulnerability detection on both source code and bytecode levels. By incorporating these cutting-edge techniques, we aim to improve the robustness and security of decentralized software systems, contributing to the ongoing efforts to mitigate the risks associated with these novel technologies.

In the field of social networks, the thesis will delve into the application of graph representation learning for link prediction tasks, focusing on criminal investigation scenarios. By leveraging the rich relational information in social network data, we aim to develop more effective methods for uncovering hidden relationships, tracking suspicious entities, and identifying potential criminal activities. Thus, the research will contribute to the broader efforts to enhance the safety and security of social networks, which play an increasingly important role in our daily lives.

In summary, this thesis aims to comprehensively explore graph representation learning techniques and their applications in security analytics for decentralized software systems and social networks. By investigating various methods, ranging from traditional graph embedding techniques to more recent deep graph learning-based approaches, we seek to demonstrate the versatility and potential of graph representation learning in addressing a wide

range of security challenges in these complex systems. Through this research, we hope to contribute to the ongoing efforts to enhance the safety and security of our digital environment for a more secure and reliable future.

1.4 Thesis Outline

This thesis is structured into several chapters, each serving a specific purpose and contributing to the overarching research goals. Below is a summary of each chapter:

- **Chapter 1: General Introduction.** This chapter sets the stage for the thesis by presenting the motivation, research objectives, and an overview of the main concepts in graph representation learning. It clarifies the potential of graph representation learning in security analytics for decentralized software systems and social networks.

- **Chapter 2: Background and Related Work.** This chapter provides an in-depth overview of the main concepts and techniques related to graph representation learning, along with an extensive literature review on security analytics for blockchain smart contracts and criminal networks. Also, it lays the foundation for network analysis using graph representation learning techniques in the subsequent chapters.

- **Chapter 3: A Database for Storing and Retrieving Blockchain-Powered Social Network Data.** This chapter introduces the design and implementation of a novel database system named SoChainDB, tailored for storing and retrieving data from blockchain-powered social networks. It discusses the challenges associated with data collection from such networks and how the proposed system addresses them. This chapter is based on our resource paper: "*SoChainDB: A Database for Storing and Retrieving Blockchain-Powered Social Network Data*" published in the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2022), July 2022.

- **Chapter 4: Link Prediction for Social Network Analysis in Criminal Investigation.** In this chapter, we explore the application of graph representation learning in link prediction tasks, specifically within the context of criminal investigations on social networks. It delves into developing and evaluating novel methods for uncovering hidden relationships between entities in criminal networks. This chapter is based on our journal paper "*Inductive and Transductive Link Prediction for Criminal Network Analysis*" published in the Journal of Computational Science, 2023.

- **Chapter 5: Multi-Level Heterogeneous Graph Embeddings for Fine-Grained Detection of Smart Contract Vulnerabilities.** This chapter proposes a multi-level heterogeneous graph embedding technique for the fine-grained detection of vulnerabilities in smart contracts. It discusses the unique security challenges of decentralized software systems

and presents the proposed method's effectiveness in addressing them. This chapter is based on our research paper: "*MANDO: Multi-Level Heterogeneous Graph Embeddings for Fine-Grained Detection of Smart Contract Vulnerabilities*" published in the 9th IEEE International Conference on Data Science and Advanced Analytics (DSAA 2022), October 2022.

- **Chapter 6: Heterogeneous Graph Transformers for Smart Contract Vulnerability Detection on Source Code and Bytecode.** This chapter delves into using heterogeneous graph transformers for vulnerability detection in both the source code and bytecode levels of smart contracts. It provides an in-depth analysis of the proposed method and its advantages over traditional techniques. This chapter is based on our research paper: "*MANDO-HGT: Heterogeneous Graph Transformers for Smart Contract Vulnerability Detection*" published in the 20th International Conference on Mining Software Repositories (MSR 2023), May 2023.

- **Chapter 7: A Tool for Vulnerability Detection for Smart Contract Source Code by Heterogeneous Graph Embeddings.** This chapter introduces a novel tool for detecting smart contract source code vulnerabilities using the proposed heterogeneous graph embedding techniques. It details the tool's implementation, functionality, and practical utility, providing a real-world application of the proposed graph representation learning techniques. This chapter is based on our tool paper: "*MANDO-GURU: Vulnerability Detection for Smart Contract Source Code By Heterogeneous Graph Embeddings*" published in the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022), November 2022.

- **Chapter 8: Conclusion and Future Work.** This final chapter offers a summary of the research presented in the thesis, discusses the implications of the findings, and provides recommendations for future research in the field. It reflects on the thesis's contributions to enhancing the safety and security of decentralized software systems and social networks.

The structure of this thesis is designed to provide a logical progression of the research, from the foundational concepts to the development of novel techniques, tools, and practical applications in the graph representation learning field.

Contribution Clarification. I conceived the original idea, established the experimental settings, verified the analytical methods, and directed the projects for Chapters 3, 5, 6, and 7. I was the main contributor in designing the technical architectures and developing the SoChainDB, MANDO, MANDO-HGT, and MANDO-GURU frameworks in Chapters 3, 5, 6, and 7. In Chapter 4, I built two graph-based machine-learning frameworks for crime link prediction and performed most evaluation experiments. Additionally, I conducted partial experiments in Chapters 3, 5, 6, and 7.

1.5 List of Publications

Throughout my doctoral studies, I have published several papers exploring various aspects of graph representation learning and its applications. Not every aspect is covered in this thesis due to space constraints. The full list of publications is as follows:

Graph Representation Learning for Criminal Network Analysis: we apply graph representation learning for criminal network analysis to establish or predict unobserved connections of entities, such as persons, organizations, and locations, to enhance investigation capabilities for large criminal cases in the publications below.

- Ahmadi, Z., **Nguyen, H. H.**, Zhang, Z., Bozhkov, D., Kudenko, D., Jofre, M., Calderoni, F., Cohen, N., & Solewicz, Y. (2023). Inductive and transductive link prediction for criminal network analysis. *Journal of Computational Science*, 102063 [13] ⋆.
- **Nguyen, H. H.**, Bozhkov, D., Ahmadi, Z., Nguyen, N. M., & Doan, T. N. (2022, July). SoChainDB: A Database for Storing and Retrieving Blockchain-Powered Social Network Data. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2022)* (pp. 3036-3045) [14] ⋆.
- Nguyen, T. H., **Nguyen, H. H.**, Ahmadi, Z., Hoang, T. A., & Doan, T. N. (2021, December). On the Impact of Dataset Size: A Twitter Classification Case Study. In *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology* (pp. 210-217) [15] †.
- Maly, K., Backfried, G., Calderoni, F., Černocký, J., Dikici, E., Fabien, M., Hořínek, J., Hughes, J., Janošík, M., Kovac, M., Motlicek, P., **Nguyen, H. H.**, Parida, S., Rohdin, J., Skácel, M., Zerr, S., Klakow, D., Zhu, D. & Krishnan, A. (2021). ROXSD: a Simulated Dataset of Communication in Organized Crime. In *ISCA Symposium on Security and Privacy in Speech Communication*, Virtual Event, 10-12 November 2021 (pp. 32-36) [16] †.
- Fabien, M., Parida, S., Motlícek, P., Zhu, D., Krishnan, A., & **Nguyen, H. H.** (2021). ROXANNE Research Platform: Automate Criminal Investigations. In *Interspeech* (pp. 962-964) [17] †.
- **Nguyen, H. H.**, Zerr, S., & Hoang, T. A. (2020, December). On Node Embedding of Uncertain Networks. In *2020 IEEE International Conference on Big Data (Big Data)* (pp. 5792-5794). IEEE [18] †.

Graph Representation Learning for Vulnerability Detection in Blockchain Smart Contracts: we apply heterogeneous graph representation learning on control-flow graphs, program call graphs, and data dependency for vulnerability detection in blockchain smart contracts in the publications below.

- **Nguyen, H. H.**, Nguyen, N.M., Xie, C., Ahmadi, Z., Kudenko, D., Doan, T. N., & Jiang, L. (2023, May). MANDO-HGT: Heterogeneous Graph Transformers for Smart Contract Vulnerability Detection. In *Proceedings of 20th International Conference on Mining Software Repositories* [19] ⋆.

- **Nguyen, H. H.**, Nguyen, N.M., Doan, H.P., Ahmadi, Z., Doan, T. N., & Jiang, L. (2022, November). MANDO-GURU: Vulnerability Detection for Smart Contract Source Code By Heterogeneous Graph Embeddings. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 1736-1740) [20] §.

- **Nguyen, H. H.**, Nguyen, N.M., Xie, C., Ahmadi, Z., Kudenko, D., Doan, T. N., & Jiang, L. (2022, October). MANDO: Multi-Level Heterogeneous Graph Embeddings for Fine-Grained Detection of Smart Contract Vulnerabilities. In *Proceedings of the 9th IEEE International Conference on Data Science and Advanced Analytics* (pp. 1-10) [21] ⋆.

Graph Similarity Learning on Multi-Target Multi-Camera Object Tracking: we build vehicle features as graph structures and customize graph similarity learning to match the vehicle objects from different cameras in the publications below.

- Nguyen, T.T., **Nguyen, H.H.**, Sartipi, M., and Fisichella, M. (2023). Multi-Vehicle Multi-Camera Tracking With Graph-Based Tracklet Features. *IEEE Transactions on Multimedia* [22] †.

- Nguyen, T.T., **Nguyen, H.H.**, Sartipi, M., and Fisichella, M. (2023). Real-Time Multi-Vehicle Multi-Camera Tracking With Graph-Based Tracklet Features. *Journal of Transportation Research Record* [23] †.

⋆: *The entire content of the publication was included in this thesis.*
§: *The partial content of the publication was included in this thesis.*
†: *The content of the publication was not included in this thesis.*

Chapter 2

Background and Related Work

2.1 Background

This section first outlines the foundational background of social network analysis, including social influence analysis, community detection, and link prediction. Then, some fundamental techniques of graph embeddings will be introduced as a basis and extended to more sophisticated graph neural networks for both homogeneous and heterogeneous graph structures. Next, we present a preliminary to introduce how blockchain-related social networks work. Finally, the latter part of the section focuses on the security challenges faced by blockchain smart contracts, emphasizing the Ethereum network.

2.1.1 Basics of Social Network Analysis

Briefly described, social network analysis is the process of uncovering hidden patterns regarding the behaviors and relations among individuals in networks through a wide range of computational and statistical methods. Examples of those patterns are the distribution of relations among the individuals, the underlying factors that determine the relations, or cohesive groups of individuals with dense relations. These methods have been applied in various domains of daily life, including economics, biology, and sociology, particularly in security and criminology. This section presents an overview of existing methods most closely related to our research in the thesis.

Social Influence Analysis

This task aims to quantify individuals' influence over others within a social network. In some contexts, it is also called influential individual identification. The influence can also be explicitly quantified to the domain or pairwise relations in open-domain settings. Within criminal analysis, where the domain is already well understood, it focuses on the overall global influence of each individual within the network. Each individual is assigned a relative importance score, as demonstrated in Figure 2.1, that measures its influence compared to others.

Inquiry into the most influential individuals in a group has been one of the leading drivers of social network analysis, especially its applications in analyzing or monitoring organized crimes. Most studies apply the centrality measures (a family of measures aiming at assessing how central, i.e., essential or influential, a node is in a network) to criminal organizations.

Typically, individuals' influence in a general social network can be measured by the following metrics:

- **Degree [25, 26]:** This metric is the number of relations or links between other individuals the targeted individual involves.
- **Closeness [25, 26]:** This metric measures how close the targeted individual is to all other individuals by aggregating the length of the shortest paths between the targeting individual and all others in the network.

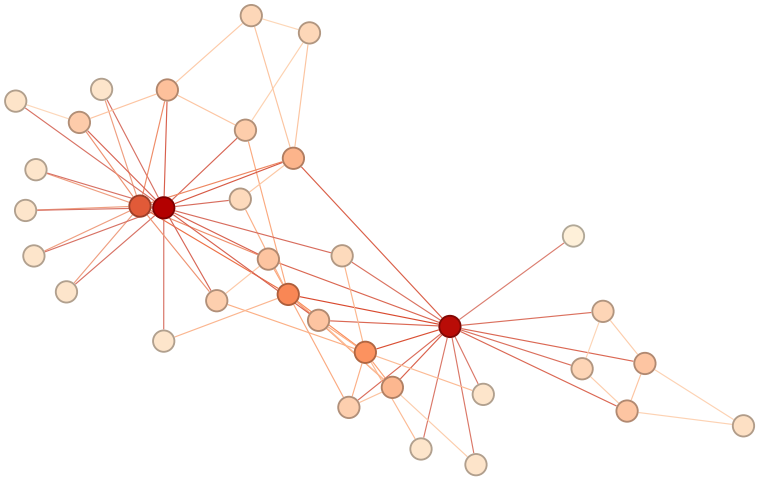


FIGURE 2.1: Example of social influence score measured for individuals in Zachary's karate club network [24].

- **Betweenness [25, 26]:** This metric quantifies the number of times the targeted individual lies on the shortest path between pairs of other individuals.
- **Pagerank score**: inspired by the ranking of webpages in search engines [27], this metric measures the importance of the targeted individual by counting the weighted vote it receives from others.
- **Authority and Hub scores:** Similar to the Pagerank score, hub and authority scores originated in search engine research [28]. These scores quantify the hub and authoritativeness of individuals in a network.

Community Detection

Frequently, individuals within a network tend to form communities. As illustrated by the example in Figure 2.2, each community is a cohesive group of individuals whose intra-community interaction is more dense and frequent than their interaction with the rest of the network. However, in most cases, this community structure is hidden as communities are not well defined, or individuals do not publicly reveal their community membership. This task aims to uncover those hidden structures using interactions among network individuals and their associated information, e.g., attributes and meta-data.

While community analysis has rapidly grown to be a popular field in general network research, its applications to security analytics still need to be completed. Research has shown that community analysis can hardly distinguish real-life subgroups in a criminal network (e.g., gangs, cliques, families, clans). For example, a study [29] inquired whether the network communities in a communication network among mafia offenders corresponded to the clan subdivision within the offenders, and the results showed that communities only partially overlapped with criminal clans in practice.

Notwithstanding these issues, investigators may benefit from community detection methods when analyzing the patterns of interaction among entities

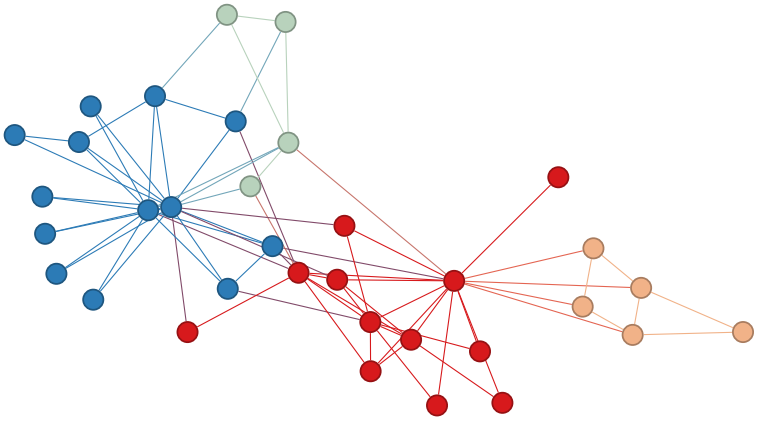


FIGURE 2.2: Example of communities of individuals in Zachary's karate club network [24]. Individuals having the same color belong to the same community.

in a criminal network. While the human eye and mind can hardly identify cohesive subgroups, clustering social relations is essential to society's (and criminal groups') organization.

Generally, existing community detection methods [30] can be categorized into four main groups, including:

- **Individual-centric methods** (e.g., K-clique-based method [31]) focus on identifying communities whose members satisfy specific properties such as the degree or reachability of other members.
- **Group-centric methods** (e.g., modularity maximization [32] and spectral clustering [33]) emphasize each community as a whole: each community has to satisfy specific properties without zooming into the individual level, such as the density of relations or interaction among community members.
- **Network-centric methods** focus on partitioning the network into several sub-networks; each is considered a community.
- **Hierarchy-centric methods** (e.g., Girvan–Newman algorithm [34]) aim to construct a hierarchical structure of communities within the network.

Link Prediction

In reality, individuals interact and communicate with each other through many channels that are not always observable. Hence, many interactions and relations among individuals in a network are hidden or not observed. Moreover, the network evolves as new interactions and connections are constantly added. This is due to the revelation of hidden interactions and relations in the past, the availability of new data sources, and primarily, individuals forming new links or interacting with unique other individuals. As illustrated by the example in Figure 2.3, this task aims to uncover these missing or hidden, unobserved interactions and relations in the network and predict the most

probable ones to be formed shortly. Specifically, given an individual u and a number k, we would like to identify top k other individuals that are not connected with u but might or would have interactions/relations with u.

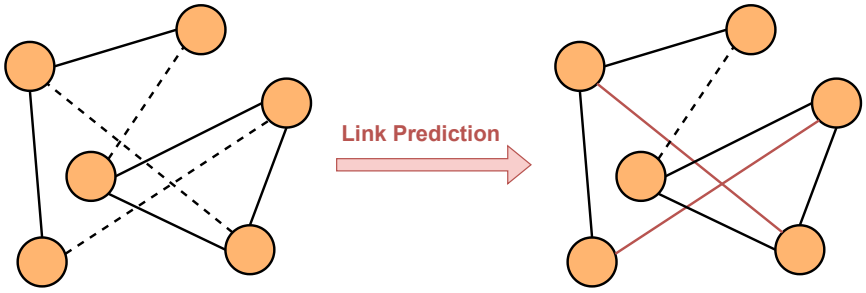


FIGURE 2.3: Example of missing links (dashed lines on the left network) and the predictions to recover them (red lines on the right network).

It is inevitable for criminal investigations to collect incomplete information about a group. Unlike the universe of suspects' interactions with other people, law enforcement agencies typically gather only a portion of these. Also, criminals have everyday lives with interactions with family, friends, partners, and colleagues. And these interactions take several forms, which law enforcement can hardly exhaustively map. Link prediction methods, increasingly popular in general social network research, can assist in identifying possible missing links. The methods may assist investigators by providing potential leads on connections not (yet) observed in the investigation, and that may request specific actions such as increased surveillance on two individuals due to the high probability of a relation between them.

In principle, all link prediction methods would, first, assign a score(u, v) to each pair of nodes (u, v) of the network and then choose the top scored pairs as predicted links for node u [35]. The score is often calculated based on measuring proximity and similarity between each u and v in the input network. Generally, existing approaches have differences in the similarity used. Some most relevant methods in social network analysis include Jaccard similarity, Adamic Adar similarity [36], Preferential attachment similarity [37], Resource allocation similarity [38], and Soundarajan-Hopcroft similarity [39]. Since link prediction is one of the significant components of our research, the differences between the five similarity methods will be described and explained in more detail in Section 4.3.1 about transductive link prediction for social network analysis in criminal investigation.

2.1.2 Graph Embedding Neural Networks

Definition 2.1.2.1 (Graph)**.** A graph can be denoted as $G = (V, E)$, where V symbolizes the set of vertices or nodes, and E represents the set of edges. The set of neighboring nodes of a vertex $v \in V$ is expressed as $N(v) = \{u \in V | (v, u) \in E\}$.

Graph neural networks (GNNs) aim to learn and create valuable and practical representations of graph structure data. Based on the rich relational

information in graph-structured data, the learning area focuses on developing algorithms capable of efficiently handling complex relational data. Various graph representation learning techniques have been developed as the discipline has advanced, ranging from fundamental graph embedding techniques to more modern tactics based on deep learning. GNNs could be categorized into two groups: Homogeneous GNNs and Heterogeneous GNNs. Although homogeneous GNNs and heterogeneous GNNs are both specialized in learning representations for graphs, they cater to different types of graph structures and apply varied methods for processing graph data.

Fundamental Graph Embeddings

Fundamental graph embeddings play a pivotal role in graph-based learning, especially before the advent of GNNs. They basically focus on converting graph-structured data into lower-dimensional spaces, where semantic relationships and graph properties are preserved. Typically, they can only handle homogeneous graphs with lower complexity than heterogeneous graphs. The traditional methods primarily leverage matrix factorization, random walks, or other heuristic methods to generate embeddings. Some widely recognized techniques include:

- **DeepWalk:** DeepWalk [5] considers short truncated random walks on the graph as a language corpus and vertices of that graph as vocabulary. It uses a skip-gram language model.
- **node2vec:** Similar to DeepWalk, node2vec [6] uses a skip-gram language model over a graph to learn its structure. In node2vec, preferences for depth-first or breadth-first sampling can be specified.
- **LINE:** The LINE embedding method [7] attempts to represent nodes as low dimensional embeddings that combine first-order (direct connection between nodes with strength weight) and second-order (direct neighborhood overlap between two nodes) proximity. The algorithm employs sampling of the second-order relationships to improve efficiency and make learning on datasets with millions of nodes and billions of edges feasible.

Homogeneous Graph Neural Networks

Homogeneous GNNs deal with graphs having nodes and edges of one single type, making them relatively less complex and more computationally efficient. They assume that all nodes share similar features and that every edge indicates the same type of relationship. Techniques like convolutional layers or attention mechanisms are common in these networks, allowing the aggregation of information from neighboring nodes and enabling the learning of representations capturing the graph's local structures and properties. Notable models in this group include:

- **Graph Convolutional Network (GCN):** GCN [8] uses a message-passing algorithm to propagate information between neighboring nodes. They typically employ multiple graph convolutional layers to learn increasingly complex features. Generally, GCN extends the capabilities of CNN by incorporating the Laplacian matrix as a first-order approximation for the propagation between the layers of spectral graph convolutions.

- **Graph Attention Network (GAT):** GAT [40] employs an attention mechanism that assigns different importance to neighbor nodes. This allows the model to efficiently aggregate information from different parts of the graph. The attention mechanism dynamically adjusts to each node's neighborhood, enabling the model to adapt to varying connectivity patterns and handle irregular structures. Their ability to capture local and global contextual information makes them particularly well-suited to complex graph data tasks.

- **GraphSAGE:** GraphSAGE [10] was introduced as an approach to formulate embeddings for unseen vertices in a dynamic network. Unlike other methods that train embeddings for every node within the network individually, GraphSAGE creates a function to generate embeddings for a node based on the local features of its neighborhoods. It first samples a node's neighbors and then employs various aggregators to refine its embedding.

Heterogeneous Graph Neural Networks

Definition 2.1.2.2 (Heterogeneous Graph)**.** A heterogeneous graph is a directed graph $G = (V, E, \phi, \psi)$, consisting of a vertex set V and an edge set E. $\phi : V \to A$ is a node-type mapping function and $\psi : E \to R$ is an edge-type mapping function. A and R denote the sets of node types and edge types, and $|A| \geq 2$ and $|R| \geq 1$.

Definition 2.1.2.3 (Metapath)**.** A metapath θ is a path in the form of $A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} \ldots \xrightarrow{R_l} A_{l+1}$, which defines a composite relation $R = R_1 \circ R_2 \circ \ldots \circ R_l$ between type A_1 and A_{l+1}, and $\circ$ denotes the composition operator on relations. Note that the **length** of θ is the number of relations in θ.

Definition 2.1.2.4 (Meta Relation)**.** A meta relation of an edge $e = (s, t)$ from a source node s to a target node t is indicated as $\langle \tau(s), \phi(e), \tau(t) \rangle$, with $\tau(s)$ and $\tau(t)$ representing the node type of s and t, respectively, and $\phi(e)$ representing for the edge type of e. A metapath can refer to a sequence of such meta relations for a sequence of connected edges.

Heterogeneous GNNs are developed for graphs containing multiple node and edge types, representing different entities and relationships. Given the diverse nature of entities and relationships in real-world graphs, heterogeneous GNNs employ advanced mechanisms for information aggregation. These may include type-specific transformation functions or attention weights that

consider the varied importance of different types of relationships, e.g., meta-relations or metapaths, to accurately capture the diversified information in such graphs. Some well-known models encompass:

- **metapath2vec:** metapath2vec [41] generates embeddings by carrying out random walks that follow user-predefined metapaths. By leveraging a heterogeneous skip-gram model, it is particularly effective in capturing the semantic relationships between diverse types of nodes, thus producing meaningful embeddings in heterogeneous networks.
- **Heterogeneous Graph Attention Network (HAN)**: Specifically designed to manage heterogeneous graphs, HAN [11] uses hierarchical attention mechanisms to understand the significance of different node types and metapaths. This model enables the learning of useful node embeddings that capture the rich contextual information in heterogeneous networks, allowing for enhanced performance in tasks such as node classification and clustering.
- **Heterogeneous Graph Transformer (HGT)**: Instead of concentrating on tokens in sentences or pixels in images or videos as the typical applications of traditional transformers, HGT [12] employs the notion of meta relations and its attention mechanisms to model different types of nodes and edges in heterogeneous graphs. By generating meta relations dynamically, HGT can effectively model complex interactions between different types of entities without relying on predefined metapaths.

2.1.3 Preliminary of Blockchain-Related Social Networks

The rapid growth of blockchain technology has attracted more and more companies and organizations working in areas related to social networks to support and integrate this technology into their platforms. Although several firms [42–44] claim that their social networks have built-in blockchain technology, most only incorporate a small part of this cutting-edge technology. For instance, one of the most straightforward approaches is often chosen to only use cryptocurrency tokens as loyalty points or a type of integrated-in currency for the users to trade in their systems through Smart Contract platforms in some of the blockchains available in the market, such as Ethereum [45], EOS.IO [46], and Binance Smart Chain [47]. The approach has the advantage of low cost and does not require advanced knowledge of blockchain technology. However, users' data is still entirely in the hands of organizations operating and developing such social networks. Therefore, these social platforms, in reality, still follow centralized models, and users do not fully control their data. At the same time, this is an essential point of a decentralized system like blockchain is aiming.

On the other hand, besides the mentioned social networks, few other platforms still focus on building and providing a completely decentralized social network, which requires a solid technical background in blockchain technology and much effort to persuade and attract users from traditional

social networks. Specifically, the most notables are the Steem network [48] and its successor, Hive blockchain [49]. The primary difference between these two blockchain-powered social networks and the rest is their completely decentralized architectures. Mainly, the core characteristics of blockchain technology, including immutability, transparency, no double spent, append-only, non-repudiation, and no single point failure, are the crucial targets in designing these social networks. These characteristics are evident through the Delegated Proof-of-Stake (DPoS) consensus mechanism using a voting system for choosing the block producers called "witnesses" per cycle and the fully decentralized designed ledger.

In addition, *Block.one*, the organization behind the popular network EOS.IO, introduced Voice being a promising decentralized social network based on the EOS.IO blockchain in 2019 [50]. The announced platform Voice inherits the advantages mentioned on Steem and Hive and also integrates *Smart Contract* system, a notable feature of EOS.IO. However, at the moment, in January 2022, the new social network is still in the beta version. Therefore, we will integrate this network into our framework after its public release.

Consensus Mechanism: As a successor to Steem, Hive applies the Delegated Proof-of-Stake (DPoS) consensus [49], which Daniel Larimer invented in 2014 as an alternative to the Proof-Of-Work consensus algorithm widely used by several famous blockchains, e.g., Bitcoin and Ethereum. The first implementation of DPoS was on a decentralized platform exchanging cryptocurrency named Bitshares in 2015 [51], then in Steem [48], ARK [52], Lisk [53], and most recently in the platforms of TRON [54], EOS.IO [46], and Hive [49]. DPoS consensus encourages blockchain users to vote and elect delegates to validate the next block as a popular evolution of the Proof-of-Stake concept. Regular users could vote on delegates through the consensus mechanism by staking their tokens into a pool and assigning those to a specific delegate. On the other hand, the delegates are responsible for achieving consensus to generate and validate new blocks. Usually, the collected rewards of the delegates are proportionally shared with their respective voters when contributing to the blockchain.

The DPoS consensus mechanism's proponents believe it is a better democratic approach for a more comprehensive and diverse group of people participating in the selection process of the next block validator. Moreover, the DPoS election system is based on the earned reputation of the delegates and not the entire wealth. Therefore, if an elected node misbehaves or does not obtain the required performance, it will be quickly suspended and substituted by another. Additionally, the limited number of validators allows the blockchain network to reach consensus more quickly. As a result, DPoS-powered blockchains are more scalable and can process more transactions per second (TPS) than Proof-of-Work and traditional Proof-of-Stake.

2.1.4 Blockchain Smart Contracts and Their Security Issues

While the first generation of blockchain was designed only to solve cryptocurrency problems, the current generation, such as Ethereum and Binance Smart

Chain, focuses on providing decentralized computing platforms [45, 47]. One new prominent feature of these trustworthy platforms is to enable *smart contract*, which can automatically execute on the blockchain and be enforced by the consensus protocol [55]. A smart contract is a *decentralized app (dapp)* with a set of rules defined by a sequence of its bytecode instructions. The smart contract can execute actions such as transferring cryptocurrency or invoking functions of other smart contracts when corresponding events happen (e.g., payment of security deposits in an escrow agreement, betting of decentralized gambling). Accordingly, smart contracts can be applied in a wide range of fields, including financial instruments (e.g., financial derivatives, crowdfunding [1]), notary (e.g., copyrights on digital arts files [2], document existence and integrity [3]) and asset tracking for the Internet-of-Things [56].

In blockchain consensus protocols, each full node of the peer-to-peer network aims to ensure the correct execution of contracts. A smart contract is executed correctly, being a necessary condition for their validity; otherwise, attackers could shuffle executions in order to divert some money from a legitimate participant to themselves. However, only the correctness of executions is insufficient to keep smart contracts secure. Like traditional software programs, smart contracts can still contain programming bugs or vulnerabilities [4] intentionally or unintentionally created by their programmers. There are numerous security issues with smart contracts [57–59], with some most popular of them include:

- *Access Control*: Failure to use function modifiers or use of `tx.origin`.
- *Arithmetic*: Bugs related to the overflow or underflow of integer.
- *Denial of Service*: time-consuming operation leads to the rejection of the smart contract.
- *Front Running*: Two dependent transactions that invoke the same contract are included in one block.
- *Reentracy*: Unexpected behavior of a contract due to reentrant function calls.
- *Time manipulation*: The timestamp of the block is manipulated by the miner.
- *Unchecked Low Level Calls*: low level functions, i.e., `call()`, `callcode()`, `delegatecall()`, or `send()` fails because of unchecked condition.

[1] https://www.ethereum.org/crowdsale

[2] https://monegraph.com

[3] https://proofofexistence.com

[4] In cybersecurity contexts, vulnerabilities mean special kinds of bugs that can be exploited and cause major security concerns. According to https://dasp.co/, all the bug types mentioned in this work can be vulnerabilities, although only a few instances of the bug types can be exploited. In this thesis, we do not need to handle the differentiation and simply treat them as synonyms.

In fact, adversaries may take advantage of undocumented methods and exploit potential bugs as well as vulnerabilities in the contracts, which can cause harm to users. Such bugs or vulnerabilities may have a more severe impact than those in traditional software as the buggy smart contracts, once deployed to a blockchain, are irreversible unless self-destructed and may lead to substantial financial losses if misused by attackers. One of the most successful attacks is "The DAO", which exploited the "reentrancy" vulnerability and managed to steal from a contract around $50M at the time of the attack [60]. Also, $31M worth of *ether* was stolen due to a critical security bug in a digital wallet contract [61].

To analyze security issues, besides traditional software engineering methods such as model checking or theorem proving, machine learning-based approaches, especially in graph representation learning, are increasingly widespread because of their compatibility and high accuracy with multiple input data sources. First, the source code or bytecode of smart contracts will be represented as an intermediate representation, such as control flow graphs, call graphs, data dependency graphs, or abstract syntax trees. The graph-structured intermediate representations have flexible abilities to capture general semantics and relationships between objects, variables, syntaxes, and functions in a software program as a smart contract. Next, based on the generated graph structures, graph neural network models are applied to *automatically* extract the input graphs' features as the node or graph embeddings used in further predicting steps. This is also a primary advantage of machine learning-based methods over conventional program analysis approaches, including model checking and theorem proving, which rely heavily on expert knowledge in defining mathematical-logic formulas, symbolic executions, and abstract interpretations. Finally, node or graph feature embeddings are utilized on specific downstream tasks such as classification, clustering, similarity, centrality, or link prediction to determine the types and corresponding locations of vulnerabilities/bugs in the input smart contracts.

2.2 Related Work

In the first part of this section, we present a list of relevant or used datasets in the experiments in the following chapters. Next, some machine learning methods applied in criminology and crime research are introduced to provide an overview of security analytics in this specialized field. In the rest of the section, we present various approaches regarding machine learning in code representation for bug detection in smart contracts. In each subsection, we also briefly mention the existing works' limitations and why our proposed methods can potentially reduce these constraints or improve the overall performance.

2.2.1 Relevant Datasets

The karate club dataset of Zachary [24] is probably the first public social network dataset. Over the years, many more public datasets have been proposed for social network studies. Baumgartner *et al.* released the Reddit dataset [62] and the Telegram dataset [63] - a social network of instant messaging apps. McAuley and Leskovec [64] also published a dataset containing user connections in Facebook, Twitter, and Google+ social networks. Cho *et al.* [65] collected the dataset from the Gowalla social network, allowing users to share their locations with friends. Yang and Leskovec [66] shared their Youtube dataset - a video-sharing type social network. Several studies use the Yelp dataset [67] to understand the impact of reviews on a location-reviewed social network. Bojer *et al.* [68] published a Kaggle network dataset, one of the largest data science social platforms. Mohri and Medina [69] released a dataset of eBay, an auction-based social network. Capocci *et al.* [70] provide a Wikipedia dataset, a social network of editors for making a global online encyclopedia. Clement *et al.* [71] publish a dataset of ArXiv - a social network of scientific collaboration among the research community. Oliver *et al.* introduce covert networks [72], a collection of past cases' networks that are collected and made publicly available. This collection includes a subset with networks among the involved individuals available along with some of their attributes (e.g., name, gender, and role). Notable examples include the networks of financial flows from the Madoff fraud case [5] and the network of terrorists involved in the 9/11 attack [6].

Unlike the basic social and criminal networks above, datasets regarding decentralized systems based on blockchain technology are more complicated and could be split into two groups:

- **Datasets of blockchain smart contracts:** which are often in the forms of source code or bytecode and require transformation to intermediate presentations (e.g., formal formulas or graph structures) before any analysis. Particularly, Smartbugs Curated [73,74] is a collection of source code of vulnerable Ethereum smart contracts organized into nine types. This dataset is one of the most used real datasets for research in automated reasoning and testing of smart contracts written in Solidity, the primary programming language on the Ethereum blockchain. It contains 143 annotated contracts having 208 tagged vulnerabilities. Smartbugs also provides a Wild collection of 47,398 unique smart contracts from the Ethereum network. SolidiFI Benchmark [75] is a synthetic dataset of vulnerable smart contracts. There are 9369 injected vulnerabilities in 350 distinct contracts, with seven different vulnerability types. Besides, Zhuang *et al.* [76], Liu *et al.* [77] and eThor [78] in their research work also introduce some datasets of smart contract source code and bytecode

[5] https://en.wikipedia.org/wiki/Madoff_investment_scandal
[6] https://en.wikipedia.org/wiki/September_11_attacks

labeled by their own rules. However, the datasets do not have fine-grained line-level labels for vulnerabilities like Smartbugs and SolidiFI Benchmark.

- **Datasets of blockchain operations:** which often contain data and history of transactions of a specified blockchain. Recently, Li *et al.* [79,80] released a dataset of operation in the blockchain-based social network Steem, called *SteemOps*. Since our SoChainDB is a database framework targeting data completeness and flexible accessibility of storing and retrieving blockchain-powered social network data, we provide in-depth analysis to show the excellence of our work over these existing ones:
  - *Data Completeness*: These works only provide a subset of the entire network. For example, Li *et al.* [80] released data with only operations of the Steem network. We argue that without other kinds of data, such as friend networks and post content, it restricts research questions that help us understand the whole picture of blockchain-based social networks. Our system is guaranteed to provide the complete data of social networks.
  - *Data Readiness and Accessibility*: The only way to use data in the current literature [79,80] is to download the archival files containing all information. Such a method has two problems: First, it does not allow users to investigate the network. For example, suppose researchers want to study an event in a specific period. They need to download and extract the entire archival file and then filter data according to the desired period. Second, archival files can produce computational problems. Since the archival files of the whole network are enormous, it is easy to exceed the computational resources. To overcome such obstacles, SochainDB provides two friendly options for users, especially data scientists and researchers, to query and download the data, including SQL-like interfaces and APIs.
  - *Data (Sub-)Instantaneity*: Archival files usually do not sync with the original social networks. Notably, the latest SteemOps dataset [80] was released on Dec 1st, 2019. It is not possible to solve emerging problems like the COVID-19 outbreak and the record price of Bitcoin with obsolete data. For this reason, the data in SochainDB is guaranteed to be near-real-time with the actual data of the Hive blockchain.

2.2.2 Crime Investigation and Machine Learning Methods

There is a growing interest in using machine learning in criminology and crime research, particularly crime prediction [81]. Most methods focus on predicting future crimes' time and/or location, while a few studies aim to find connections and links among the crimes. In this section, we review the

models based on machine learning to deal with the crime linkage problem and general crime prediction.

Crime Linkage

Crime linkage studies are generally based on the similarity of criminal behavior, which relies on three assumptions [82, 83]: (1) criminal behavior is consistent, that is, the same offender will behave similarly over time, (2) different criminals show distinctive criminal behaviors that are different from each other, and (3) criminal behavior is measurable through a direct relationship and homology between the characteristics of offenders and their behavior through quantitative models.

The crime linkage problem can be seen as a binary classification task that aims to find serial crimes committed by the same offenders. In this setting, it is evaluated whether each of all possible crime pairs represents a serial crime pair or not. Due to insufficient evidence, it is sometimes difficult to determine if a crime is serial, so these two-way decisions are prone to error. A recent study attempted to model the problem with a three-way decision, where the data space is divided into three possible regions (i.e., positive, negative, and boundary) based on two thresholds, so those samples that are difficult to distinguish given the existing information are placed in the boundary area [84]. Then, they tried to automatically learn the thresholds of the three-way decisions without the need to establish explicit loss functions. Nonetheless, serial crime pairs are much less common in many real-world cases than non-serial crime pairs. To address this challenge, some studies applied class imbalance algorithms [85]. In a particular robbery case, they focus on the indistinguishable case pairs at the classification boundary rather than resampling smaller or larger classes to handle the imbalanced dataset. Chi *et al.* also presented a decision system that determined if two robberies belong to the same series of cases [86]. However, their system is quite limited in that it considers only robbery cases and requires police officers to manually mark each case's characteristics.

Due to the increasing attention to deep learning methods and their promising results in various applications, a recent study uses an adaptive deep Q-learning network with reinforcement learning to develop a robust crime prediction model [87]. In another work, Wang *et al.* extended the ResNet model [88] for spatiotemporal crime forecasting [89]. A crime forecasting system using an attention-based sequence-to-sequence model and convolutional variational autoencoders has also been proposed [90]. Previously, Simonyan and Zisserman presented a two-stream deep learning approach that learned video representations by dividing video streams into two components, one representing a spatial stream and the other a temporal one [91]. They used a Convolutional Neural Network (CNN) architecture to identify spatial dependencies, further enhanced by a Long-Short Term Memory (LSTM) that captured temporal patterns. The joint use of CNNs and LSTMs showed highly complementary behavior in capturing spatial and temporal features for video classification [92]. Later, Solomon *et al.* proposed machine learning-based

approaches for crime linkage to parameterize crimes with spatiotemporal information and automatic and manual language features extracted from police reports [93]. In doing so, they used two types of samples: positive pairs, which were pairs of burglaries committed by the same criminal, and negative pairs, which were pairs of burglaries committed by different criminals. Then, the model predicted crimes committed by new criminals that were not observed in the training phase. Ghazvini *et al.* also proposed a model that detected serial crimes by isolating short-term repetitiveness using neural networks [94].

All related work presented so far relates to transductive link prediction, where approaches have aimed to build new connections between known offenders. Inductive link prediction, by contrast, focuses on cases where new offenders are added to the scope. Expanding the analysis beyond the study of crime, Bojchevski and Günnemann [95] described network nodes using a Gaussian distribution instead of a simple low-dimensional vector used in previous studies (e.g., [6]). Then, a dissimilarity measure with respect to the Gaussian distribution was defined to minimize the heterogeneity of adjacent nodes: each time a new node was added to the network, the approach would predict its links to known nodes while minimizing the dissimilarity between the connected ones.

Crime Prediction

Crime prediction methods in the literature have largely ignored the role of co-offending in committing a crime. Instead, they have focused on modeling observed crimes spatially and temporally to predict the time and location of future crimes. Crime prediction methods can be divided into two categories: traditional empirical methods and spatiotemporal methods. Spatiotemporal models, including time series [96] and Kernel Density Estimation (KDE) [97], are commonly applied for crime prediction and are related to the crime linkage problem. Variously, traditional methods consider time and location independently [98] and aim to focus on predicting crime hotspots and crime risk areas [99]. These methods are less relevant to our study as they ignore the connection between the crimes. However, we provide a brief overview of this literature to review the machine learning methods used in this field.

As for early studies, Olligschlaeger examined using Multi-Layer Perceptron (MLP) in GIS systems to predict drug-related calls at 911 call centers in Pittsburgh, USA [100]. He trained a simple MLP architecture with only nine neurons and a single hidden layer on a dataset with three collected features indicating the number of calls received in each map cell area related to weapons, robberies, and assaults. Later, Gorr *et al.* compared different regression approaches to predict a set of crime categories using Pittsburgh data [101]. They ran regression functions of different complexity on the same data and found that more sophisticated methods outperformed simple time series. In particular, they found that the predicted mean absolute error was improved through a smoothing coefficient, that is, by applying more weight to recent data. However, results from the Prophet model [102] applied to

crime occurrence datasets from three major US cities showed that time series models could outperform neural networks.

SVMs have shown successful results in various applications, including hotspot location prediction [103]. Yu *et al.* compared the competency of SVMs to other well-established machine learning approaches like Naive Bayes and Random Forests [104]. They found supportive evidence aligned with the theory of the recurrence of residential burglary at a particular place. In some types of crimes, such as burglaries, serial offenders remain in a certain area due to familiarity with the region, even though the proximity to their home could compromise their anonymity. Thus, their incidents are concentrated within a ring-shaped area centered on an outbreak point [105]. Furthermore, Mohler *et al.* [106] and Ratcliffe [107] proposed two different temporal crime modeling approaches and gained insights that validated the well-known claim that offenses can be driven by the availability of opportunities.

Many of the methods in the literature rely solely on spatial dimensions of the incidents, including KDE [108]. Originally, Silverman used KDE to divide the study area into grids of regular cells and estimated a density value for each cell via a kernel function that estimates the probability density of crime incidents [109]. By extending KDE to employ space and time variants, Nakaya and Yano proposed a crime cluster analysis with a temporal dimension that can simultaneously visualize the geographical extent and duration of criminal clusters [97]. Going one step further, Toole *et al.* used criminal records to identify spatiotemporal patterns at multiple scales [110]. They employed various quantitative tools to identify significant correlations across both space and time in the behavioral crime data.

2.2.3 Code Representation and Machine Learning Techniques for Bug Detection in Smart Contracts

We discuss the main differences between our work and closely related work on smart contract bug detection.

Conventional Bug Detection Techniques

Bug and vulnerability detection has been an important research problem for a long time in various domains of computer science, including programming languages, systems, cybersecurity, and software engineering. Furthermore, with the rise of more diverse and complex languages and software systems, more general, scalable, and accurate bug/vulnerability detection and prevention are essential. On the other hand, smart contracts in blockchain technology have received increasing popularity and reputation, and hence, vulnerability detection for smart contracts is becoming a popular topic [58, 111]. Many studies detect specific types of bugs or vulnerabilities using conventional program analysis and software engineering and security techniques, such as testing/fuzzing [112–116], symbolic execution [117–121], and static/dynamic program analysis [78, 122–126]. For example, OYENTE [127] uses symbolic execution to explore execution paths in smart contracts as much as possible

and search for four types of bugs. SmartCheck [128] uses static analysis techniques to check smart contract code for patterns that match pre-defined rules about vulnerabilities and code smells. Several other studies [129–135] use formal verification to check smart contracts' safety and functional correctness according to certain human-defined specifications [136]. However, in contrast to our automatic bug pattern detection method, such security analysis techniques are built to discover specific vulnerabilities according to manually defined patterns or specifications. They often need customized implementation of the testing, analysis, and verification algorithms for the specific smart contract language and vulnerability types; their analysis algorithms can be very different for source code and bytecode, limiting their flexibility for new languages or vulnerability types. Although our learning-based approaches also require customized front-end code parsing and control-flow graph constructions, our graph-learning components in MANDO, MANDO-GURU, and MANDO-HGT are independent of the languages and can be applicable to new vulnerability types.

Learning-Based Bug Detection Techniques

Many studies are based on machine learning and deep learning for detecting bugs in either source, bytecode, or binary. Generally, there are more deep learning-based techniques for source code than bytecode/binary.

Learning-based bug detection for general source code. Software programs have explored learning from heterogeneous graphs for vulnerability detection, code search, and other tasks. Most previous studies either consider token sequences of code, data flow graphs, or control flow graphs for individual functions separately or consider cross-function relations independently from the within-function relations due to scalability and design constraints in their methods. For example, VulDeePecker [137] uses both syntax structures and dependency slices to represent programs and employ commonly used neural network models to learn the programs' embedding and identify vulnerability patterns for C/C++ programs. VulDeeLocator [138] extends the work by adding attention-based granularity refinement to identify fine-grained line-level vulnerability locations. BGNN4VD [139] also uses combined code representations in the abstract syntax trees and control- and data-flow graphs to learn vulnerability patterns via bilateral graph neural networks for C/C++ programs. Some other studies consider different kinds of code representations and learning techniques for source code in various languages [140–147] and very few consider using heterogeneous graphs for source code representation [148]. However, the existing approaches are almost designed for other languages and unsuitable for Solidity smart contracts.

Our MANDO, MANDO-GURU, and MANDO-HGT are the first to consider heterogeneous graph learning that combines control-flow graphs and call graphs and employs the learned embedding for recognizing vulnerabilities in Solidity smart contracts. Different from all previous studies, our frameworks based on heterogeneous contract graphs are able to represent a smart contract's syntactic and semantic information more comprehensively,

combining fine-grained individual statements and lines of code within each function with cross-function call relations together for pattern learning and search. In addition, they can be scalable and generalizable to different types of vulnerabilities.

Learning-based bug detection in smart contract bytecode and general binary code. DC-Hunter [149, 150] uses an unsupervised graph embedding algorithm to encode normalized and sliced code graphs into comparable vectors so that vulnerable smart contracts can be identified when compared to the vectors of known vulnerable ones. For their high precision, they need to heuristically normalize instructions and data in the bytecode generated by different compilers and slice the code following contract-specific data- and control- flows during simulated executions. L-GCN [151] splices a graph convolutional network for control-flow graphs together with a long short-term memory network for segmented opcode sequences of bytecode and trains the spliced networks to classify vulnerable bytecode. Zhu *et al.* [152] learn bytecode but for a different problem of finding similar smart contracts. Also, among the limited literature on deep learning-based vulnerability detection methods in smart contracts bytecode [153–157], some use control-flow and data-flow graphs. However, they still use homogeneous graph learning techniques, while our approach customizes heterogeneous graph learning for both source code and bytecode of smart contracts. Besides smart contract bytecode, a few studies have adopted deep learning techniques for general binary code [158–162]. They also have not considered heterogeneous graph learning techniques either, although some of those learning techniques can be combined with ours.

Our MANDO-HGT framework is different from those studies in that we adapt supervised graph learning directly to the bytecode control-flow graphs without the need for heuristic code normalization or slicing, and we are able to perform fine-grained function-level vulnerability detection. MANDO-HGT uses a triplet network composed of sequence transformers to recognize similar/dissimilar smart contracts without source code, even when different optimization options and compiler versions produce their bytecode. We focus on a different problem of vulnerability detection in smart contract bytecode using graph-based learning while handling only one compiler version and optimization. It would be useful to consider extending our graph-based approach to handle different compiler versions and optimizations.

Graph Embedding Neural Network Techniques

A few studies have detected smart contract vulnerabilities using graph neural network-based embedding techniques. Zhuang *et al.* [76] represent each function's syntactic and semantic structures in smart contracts as a contract graph and propose a degree-free graph convolutional neural network with expert patterns to learn the normalized graphs for vulnerability detection. They also provide more interpretable weights by extracting vulnerability-specific expert patterns for encoding graphs [163]. In the Peculiar tool [164], Wu *et al.* present a pretraining technique based on customized data flow graphs

of smart contract functions to identify reentrance vulnerabilities. However, their methods face various limitations: Relying on expert patterns, their graph generator only works with some pre-defined Major and Secondary functions before generating the contract graphs, leading to poor performance in the graph generation process compared to our MANDO, MANDO-GURU, and MANDO-HGT frameworks. Besides, pre-defined patterns also restrict them to detect only two specified bugs, Reentrancy, and Time Manipulation, in Solidity source code. In contrast, the heterogeneous graph structure allows our frameworks to be more general and flexible in exploring different vulnerability types without requiring any pre-definitions. Other studies use other forms of embeddings in code fragments or the graph/tree structures generated. For example, SmartEmbed [165] employs serialized structured syntax trees to train word2vec and fastText models to recognize vulnerabilities. SmartConDetect [166] treats code fragments as unique sequences of tokens and uses a pre-trained BERT model to identify vulnerable patterns. Meanwhile, Zhao *et al.* [167] use word embedding together with similarity detection and Generative Adversarial Networks (GAN) to detect reentrance vulnerabilities dynamically.

Although the existing deep/graph neural network-based techniques alleviate the problem by automatically learning bug patterns from certain representations of existing code, such as syntax trees and data-/control-dependency graphs, they have treated the trees/graphs as flattened sequences or conventional graphs disjointing each other and have *not* utilized particular kinds of control flow and call relations in the contract code to capture their semantics more comprehensively. Moreover, they often treat nodes and edges in the tree- and graph-representations of source code *homogeneously*, ignoring fine-grained differences in their types and locations. As a result, they could only search for coarse-grained whole-graph-level smart contract vulnerabilities, which are not accurate enough to locate the line-level locations of vulnerabilities. Different from such existing techniques, our unique graph encodings in MANDO, MANDO-GURU, and MANDO-HGT can accurately capture vulnerability patterns and locate fine-grained vulnerabilities at the line level.

Chapter 3

A Database for Storing and Retrieving Blockchain-Powered Social Network Data

Social networks have become an inseparable part of human activities. Most existing social networks follow a centralized system model, which despite storing valuable information of users, arise many critical concerns such as content ownership and overcommercialization. Recently, decentralized social networks, built primarily on blockchain technology, have been proposed as a substitution to eliminate these concerns. Since decentralized architectures are mature enough to be on par with centralized ones, decentralized social networks are becoming more and more popular. Decentralized social networks can offer both common options like writing posts and comments and more advanced options such as reward systems and voting mechanisms. They provide rich ecosystems for influencers to interact with their followers and other users via staking systems based on cryptocurrency tokens. The vast and valuable data of the decentralized social networks open several new directions for the research community to extend human behavior knowledge. However, accessing and collecting data from these social networks is not easy because it requires strong blockchain knowledge, which is not the main focus of computer science and social science researchers. Hence, in this chapter, we propose the SoChainDB framework that facilitates obtaining data from these new social networks. To show the capacity and strength of SoChainDB, we crawl and publish Hive data - one of the largest blockchain-based social networks. We conduct extensive analyses to understand the insight of Hive data and discuss some interesting applications, e.g., games, and non-fungible tokens market built upon Hive. It is worth mentioning that our framework is well-adaptable to other blockchain social networks with minimal modification.

3.1 Introduction

Social networks provide many useful services for their end-users; therefore, they are a part of billions of users' lives worldwide nowadays. For example, Facebook had around 2.85 billion monthly active users at the end of March 2021 [168], and Twitter has nearly 300 million active users [169]. Such a large userbase creates a rich and colossal dataset of various aspects of human activities. However, unfortunately, most social network services are deployed upon a centralized architecture. In other words, each social network is under the umbrella of a particular organization or company. Despite having many advantages, a centralized architecture still contains several fundamental disadvantages:

1. The content ownership is not in the hands of its creators. Although users generate content through their interactions in social networks, the content is hosted by the service providers. Hence, "who is the actual

owner?" is still an open question. The data leakage risk makes the problem worse.

2. Internet censorship is another thread of centralized architectures. Service providers are under pressure from other organizations to remove or delete posts or comments displeasing those organizations. Therefore, they are ineffective tools to protect the voice of their users.

3. Due to their large user base, social networks are being exploited for commercialization by marketing companies and advertising agencies. Advertising content is increasingly integrated into many popular social media platforms without permission from users. Such activities could harm the engagement of users' experience, and have a negative impact on users' behavior.

Due to these drawbacks, distributed social network architecture has been proposed as a reasonable substitution. The technology behind it is blockchain which ensures the completeness of data by leveraging cryptography. Since its proposal, social networks powered by blockchain have been evolving gradually, and they are now ready to serve millions of users. Decentralized social networks provide many benefits: First, their functionality is on par with conventional social network platforms. For instance, standard features such as following users, posting articles, and writing comments are all available in blockchain-based social networks. Second, they inherit transparency from blockchain technology. Since all data is stored via blockchain, it is publicly accessible, and decentralization makes blockchain social networks impossible to manipulate. Third, users of blockchain social networks are rewarded for their activities. Such a system encourages users to engage more in social networks and benefits their activities. Finally, we can leverage blockchain technology of decentralized social networks to build applications such as games or trading platforms to help the existing network user base. These potentials in decentralized social networks offer the research community many novel directions to understand human behaviors through valuable and massive data generated by a variety of user interactions on the system. However, collecting data from decentralized social networks has several challenges: First, blockchain knowledge is required, which is not negligible and could create a high barrier for scientists from other fields e.g. social science who are not knowledgeable in cryptography. Second, the demand for a computational resource to synchronize complete data is usually high and expensive. For example, we required a server with more than 100GB RAM to synchronize one full node of the Hive network. Third, a cleaning process is a must since blockchain can be used for multiple purposes, not only social networks. Therefore, we propose SoChainDB, a framework for crawling data from decentralized social networks, and publish Hive's data - one of the largest decentralized social networks built on blockchain technology. The contribution of our work can be summarized as follows:

- We first propose SoChainDB, a generalized database framework and publicly available pipeline for extracting data from blockchain-based decentralized social networks.
- We publish the entire dataset of one of the largest blockchain-based social networks called Hive. It is available to download via multiple methods such as public API services and compressed archive files.
- We provide several unique use cases of blockchain-based social networks that could be potential future directions for the research community to explore.

Ethics Declaration: Data stored in the Hive blockchain cannot be manipulated by any individuals or organizations, and there is no restriction in accessing data. Therefore, the Hive blockchain data, including comments, votes, posts, and all other blockchain-type transactions, are considered public data. Accordingly, no permission is required to collect, store, analyze, and publish the data. The data stored in our SochainDB is marginally different from the stored data in Hive due to our noise filtering preprocessing step.

SoChainDB is publicly accessible at `http://sochaindb.com` and the dataset is available under the CC BY-SA 4.0 license.

3.2 Overview of Hive Blockchain

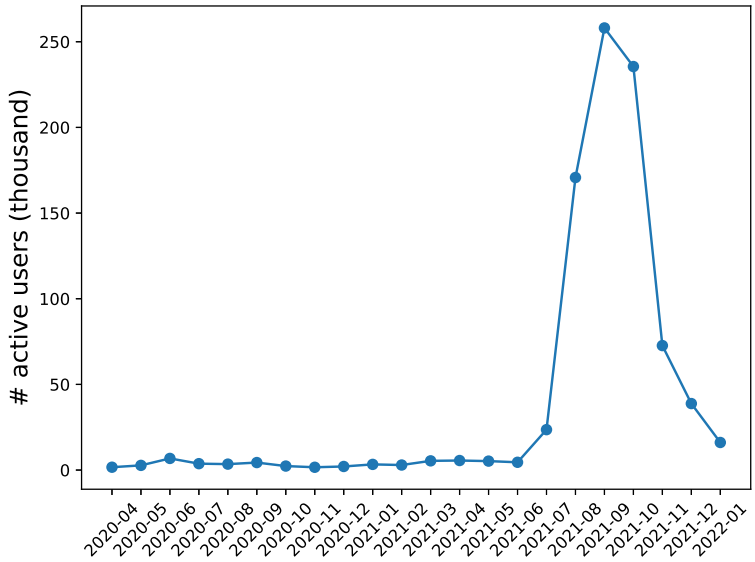


FIGURE 3.1: Growth of active users on Hive over time. From April 2020 to May 2021, the monthly growth of active users is stable, around 1,600 to 6,700 users per month. Still, since June 2021, its monthly growth has increased significantly because of the peak price of Hive tokens during this period [170].

After a hard-fork named v0.23.0 from Steem network on March 20, 2020, which is related to some conflicts between Steem core communities and the

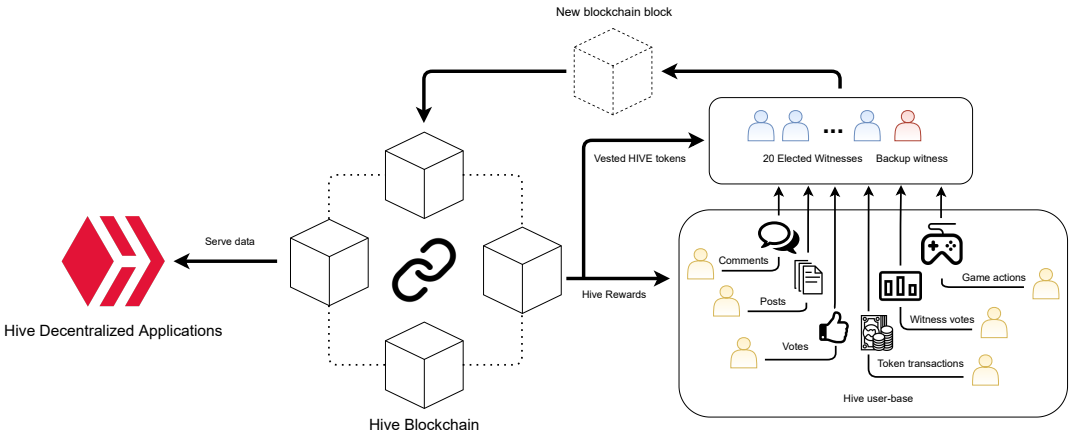


FIGURE 3.2: Overview architecture of Hive blockchain-based social network.

new board of Steemit Inc. [171], Hive became an entirely community-based blockchain. In more than a year, the growth of active users per month in Hive blockchain has remarkably increased from approximately 1,600 active users in April 2020 to more than 250,000 in August 2021 (see Figure 3.1)[1]. This is partly due to the transformation from the Steem communities to the Hive network, besides the fact that the decentralized social network concepts have become more and more widespread over time. In the current version of the SoChainDB framework, we only focus on supporting and integrating the data extracted from the Hive blockchain to ensure the overall best performance. Moreover, Hive and Steem share a nearly similar architecture; therefore, our approach could easily and quickly adapt to the Steem network with minimal modification in the subsequent versions. Figure 3.2 shows the overview architecture of Hive blockchain. Here, we only discuss the core components of Hive blockchain used for generating a decentralized social network.

Hive - Social Network: Since Hive supports all basic functionalities of traditional social networks, users can create, edit, comment, and share posts. The posts can contain text, links, hashtags, mentions, and metadata such as timestamp, author, edit the information. Note that multimedia data, including images or videos, are not stored directly in the Hive blockchain due to the block size limitations and the blockchain's overall performance, especially when a new block is created and broadcasted throughout the network in seconds. Moreover, other users can view and interact with the post by reply, comment, and reblog. These ensure users' consistent experience and similarity to regular social networks.

Hive - Blockchain: The Hive underlying blockchain architecture allows easy storage and retrieval of immutable chains of large amounts of data and information. It also provides an efficient transaction platform in only three seconds without any fee. Transaction confirmation time and fees are usually among the most important challenges of promoting a blockchain's development and adaptability of use. For example, the Bitcoin network takes

[1]The users are classified as "active" if they publish at least one post, comment, or vote, even if they do not have any actions later.

an average of ten minutes to validate a new block with transaction fees that tab to 60 USD at a price in April 2021 [172]. Besides, when Hive witnesses generate a new block, it includes all verified transactions or operations that users perform. These operations could be classified into four primary groups [48,49]: (i) post and vote, (ii) witness election, (iii) followers/followings, and (iv) cryptocurrency transfer.

Hive - Tokens and Rewards System: Similar to the miners of Bitcoin and Ethereum, the Hive witnesses receive Hive cryptocurrency tokens rewarded by Hive blockchain when generating and validating new blocks. The Hive tokens can power up a Hive account for more substantial voting power and increased curation rewards, more resource credits to make transactions on Hive blockchain, and more stake in Hive governance to assign and vote witnesses and projects. Also, the Hive platform provides another unique reward system based on an upvote and downvote mechanism. It is integrated into the blockchain core using Hive tokens, and Hive Backed Dollars (HDB) tokens. Those authors who write engaging and trending content can receive Hive tokens or HDB tokens from other users on the network. Moreover, some Hive-based back-ends such as *PeakD* [173] or *Hive.blog* [174] can rank a post based on users' interaction and the number of Hive tokens staked. The higher the rank, the more likely the post would appear on these decentralized web applications' front page or trending tabs.

3.3 Dataset Collections & API Service

3.3.1 Pipeline

Figure 3.3 illustrates the pipeline of SoChainDB, including the following four steps:

- *Scraping:* To obtain the entire Hive blockchain dataset, we should synchronize a Hive full-node. Setting up a full node often requires advanced blockchain knowledge of custom configuration to select the suitable blockchain plugins and index the data in a sufficient time with reasonable computational resources. These settings are also relevant to the peer-to-peer protocol that synchronizes the data block. Thus, if a regular researcher or a data scientist wants to access and exploit information from these types of decentralized social networks, these are significant obstacles. Moreover, ensuring data integrity is always the highest priority of our framework when extracting from each block on the Hive distributed ledger. Hence, all scraped data are re-checked with public peer nodes for completeness, accuracy, and consistency.

- *Cleaning:* Our system encodes the data into the NDJSON-formatted files, in which every line has JSON format, and then splits them into chunks as an intermediate step for simultaneous preprocessing. This technical step is necessary because the size of the uncompressed blockchain dataset is almost one terabyte. Moreover, the Hive blockchain provides several

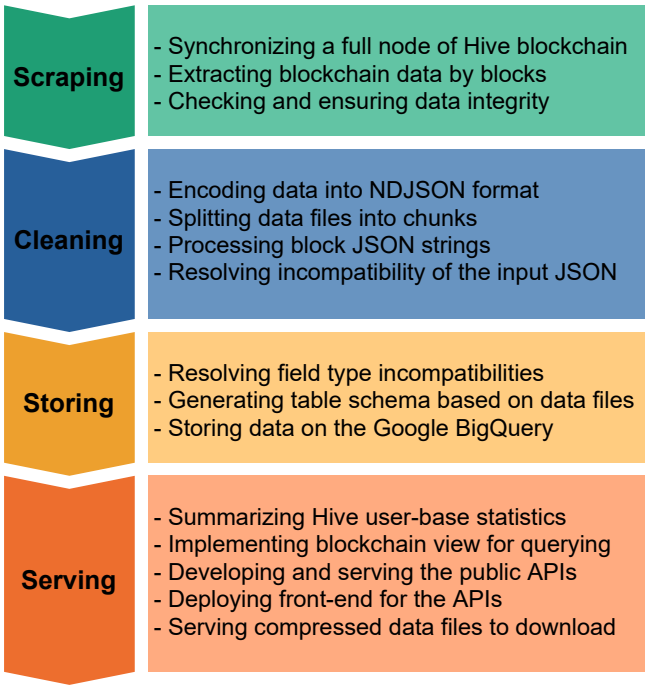


FIGURE 3.3: SoChainDB general pipeline.

dynamic JSON fields that allow users to push data in custom fields encoded as a regular string. Thus, the custom JSON strings should be decoded, and all field incompatibility issues should be refined and resolved before storing in a cloud service.

- *Storing:* In this step, table schemas based on the data chunks are automatically generated with the types of each block field being homogeneous before uploading them to a suitable cloud platform specialized in big data. In the current version, we use Google BigQuery as the central platform to store the decentralized social network data because of its high adaptability and compatibility with multiple varieties of data, as well as the high performance on querying big data and scalability [175]. However, our system architecture allows us to migrate to other platforms with similar functionalities to Google BigQuery flexibly with little change.

- *Serving:* We have implemented several blockchain views to optimize the query time and secure our APIs from anomaly users' behaviors while crawling data. As significant and core features, our public APIs are listed and served through our front-end at `http://sochaindb.com`. The detail of these APIs is described in Section 3.3.2. Likewise, we summarize blockchain operation statistics and present some prominent analyses about the Hive decentralized social network in Sections 3.4.2 and 3.4.2. We provide an HTTP service to download the compressed archive files at `http://sochaindb.l3s.uni-hannover.de`.

3.3.2 SoChainDB's Public APIs and Homepage

APIs service: SoChainDB provides a RESTful API service built on Falcon [176] to query our clouded data on Google BigQuery through some endpoints. This highly optimized framework has significant features, such as asynchronous I/O support and simple API modeling. Falcon also showed an outstanding performance via intensive experiments on benchmarks and comparison with various other Python web API frameworks in several realistic scenarios in 2018 [177]. It assists us in accelerating the incoming requests to access our database in parallel versus sequential ways. Our APIs are generally designed to process big datasets with thousands of requests per second. The API service is deployed at `http://sochaindb.com/hive-api/v1.0.0/`, and its source code is publicly accessible at our Github repository [2]. Since the hard-fork of Hive from Steem happened in late March 2020 [171], our APIs in the early version could only support the Hive blockchain data from March 27, 2020 to December 6, 2021. We schedule to update the database every month and add the Steem blockchain data in the subsequent versions. All our RESTful APIs use the GET methods divided into three groups to meet the basic requirements of datasets that suit social network researchers and data scientists:

- *Blocks*: could be used to crawl the entire blocks containing all of the transactions from the Hive blockchain data. This "blocks" API allows users to collect complete data of each block in the public ledger, including a large amount of information about various operations types.
- *Posts*: could be employed to crawl posts data that we filtered from the blocks transactions. In general, the collected posts are the transactions containing operation type as *comment_options_operation* having a title field.
- *Comments*: could be used to get comments that we filtered from the blocks transactions. The comments transactions have the same post-operation type *comment_options_operation* with an empty string in title.

We also provide APIs specialized in collecting data for statistical purposes with a 10,000 default size. However, users can easily modify the size by changing the *size* parameter before requests. For example, through the GET request, we can:

1. Crawl a list of users having the top amount of posts: `http://sochaindb.com/hive-api/v1.0.0/top_posts?size=1000`
2. Crawl a list of users having the top amount of comments: `http://sochaindb.com/hive-api/v1.0.0/top_comments?size=1000`
3. Crawl a list of contents for top posts and comments: `http://sochaindb.com/hive-api/v1.0.0/top_words?size=1000`

[2] `https://github.com/SOCHAINDB/hive-db`

Parameter	Description	Default	Accepted Values	APIs			
				blocks	posts	comments	statistics
size	Limit the results size of a request. A data sample might be large, especially the block samples. Users can set size for reducing runtime.	25	Integer	✓	✓	✓	✓
fields	Get fields in the schema. Not all fields are useful, and it depends on individuals' purposes. Users can add a list of fields for reducing runtime.	"*"	String; List of strings separated by comma	✓	✓	✓	
witnesses	Filter data by a "witness" or a list of "witnesses". It is sometimes essential information for analyzing.	None	String; List of strings separated by comma	✓	✓	✓	
ids	Filter data by the identified blocks IDs.	None	String; List of strings separated by comma	✓	✓	✓	
block_ids	Filter data by the blocks hash, which is similar to IDs, however, this is used to reference each block in the database.	None	String; List of strings separated by comma	✓	✓	✓	
operations	Filter by the operation types of the transactions in the blocks.	None	String; List of strings separated by comma	✓			
after	Filter data after a specified time. The first available time in our database is at 16:40:09 UTC on 27th March 2020 for the current version.	None	UTC format or timestamp	✓	✓	✓	
before	Filter data before a specified time. The last available time in our database is at 23:59:57 UTC on January 31st, 2022 for the current version.	None	UTC format or timestamp	✓	✓	✓	
authors	Filter by the authors. If users are interested in some posts or comments, they can add a list of authors to search for more actions.	None	String; List of strings separated by comma		✓	✓	
permlinks	Filter by "permlink" being a partition of posts or comments' URL on Hive social network. Users can add a list of "permlinks" for reducing runtime.	None	String; List of strings separated by comma		✓	✓	
post_permlinks	Filter the comments in the posts having the "permlinks."	None	String; List of strings separated by comma			✓	
words	Filter the posts or comments which contain the specified input words. This could help users catch some social network trends by searching the hot trending words.	None	String; List of strings separated by comma		✓	✓	
tags	Filter the posts which contain the specified hashtags. This might help users search the posts more accurately than the words parameter.	None	String; List of strings separated by comma		✓		

TABLE 3.1: SoChainDB API parameters. The details of parameters *fields* and *operations* can be found in the two respective links: https://github.com/SOCHAINDB/hive-db/blob/master/assets/fields.md and https://github.com/SOCHAINDB/hive-db/blob/master/assets/summary.org/#operation-types.

In Table 3.1, we summarize the list of API parameters used in the APIs service and their compatibility for each provided API.

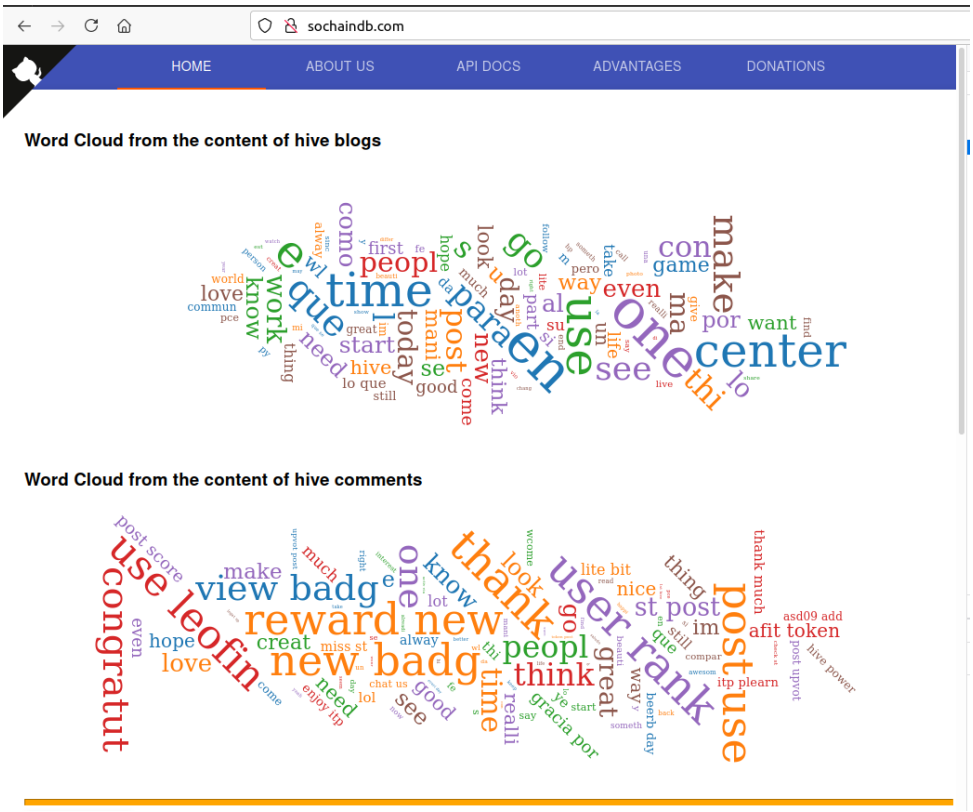


FIGURE 3.4: SoChainDB homepage.

SoChainDB Homepage: We built a website to show the demos and update more information. Figure 3.4 is a screenshot of the SoChainDB homepage. The top left corner of our homepage links to our Github repository. The navigation bar on top of the website provides several valuable resources to explore our website. For instance, the "API Docs" menu shows a detailed document of our provided APIs with various examples for ease of use. The rest of our homepage visualizes some notable statistics of our dataset, e.g., word cloud of top comments and posts, top active users of Hive networks according to the number of posts and comments.

3.4 Use Cases

In this section, we present some potential SoChainDB use cases through three different analyses: (i) Hive social network; (ii) Splinterlands - a card game stored in Hive; and (iii) NFTShowroom - an NFT marketplace built on top of Hive. The SQL queries [3] incorporated to extract all data described in this section from SoChainDB, and some sample data of our APIs [4] can be found in

[3] https://github.com/SOCHAINDB/hive-db/blob/master/assets/queries.md
[4] https://github.com/SOCHAINDB/hive-db/tree/master/sample_data

our Github repository. We first introduce the overview of the Hive ecosystem and then present an in-depth analysis of the use cases.

3.4.1 Hive Ecosystem Overview

Hive is first used for its social network platform but with Hive Engine, a side-chain layer enabling smart contracts to work on its network, Hive also allows developers to build various decentralized applications on top of its blockchain. There are several applications built on the Hive ecosystem. Some famous ones are: (i) *Game*: Splinterlands, a multiplayer magic card game, and Rabona, a soccer management game; (ii) *Social*: LeoFinance, a crypto traders community, and Actifit, a community for users to share their workouts; (iii) *Non-Fungible Tokens Markets*: NFTMart and NFTShowroom as notable marketplaces to purchase digital assets whose owners can have proof of ownership; (iv) *DeFi*: The platforms to raise funds for cryptocurrency for any project. The most famous decentralized application is DLease; and (v) *Video*: Vimm and 3Speak are two most well-known Hive blockchain-based video-sharing services.

3.4.2 Analysis of Hive Social Network

Overall Analysis

This section analyzes various aspects of the Hive social network to depict Hive users' rich and massive data. Table 3.2 shows the overview statistics of the social network. The table shows that Hive social network obtains a sizable adoption. For instance, the total number of active users is more than 760,000, and around 1,200 new users join the network every day. These users generate more than three million posts in total and more than five thousand posts per day. To further illustrate the evolution of the network, we plot the growth of active users over time in Figure 3.1. The figure shows that the growth of active users has significantly increased by approximately 15,500% in over a year. It is an important signal to show that the Hive social network is an active platform that receives more and more users' attention. We further plot the Hive users' activities over time in Figure 3.5 through the number of posts, comments and communities.

The figure depicts the regular activities of Hive users every month. Specifically, these users' posts and comments continuously increased from August 2021 and reached 240K and 700K respectively in January 2022. It shows another evidence that Hive is steadily becoming a platform for users to share their daily life habits.

To form a network, each user has an option to follow others. The essential feature of social networks and the follower/following network should follow a power-law distribution [178]. Figure 3.6 shows the follower/following distribution of users in the Hive network.

The figure states that Hive social network's follower/following relationship follows a power-law distribution, and it is similar to the centralized

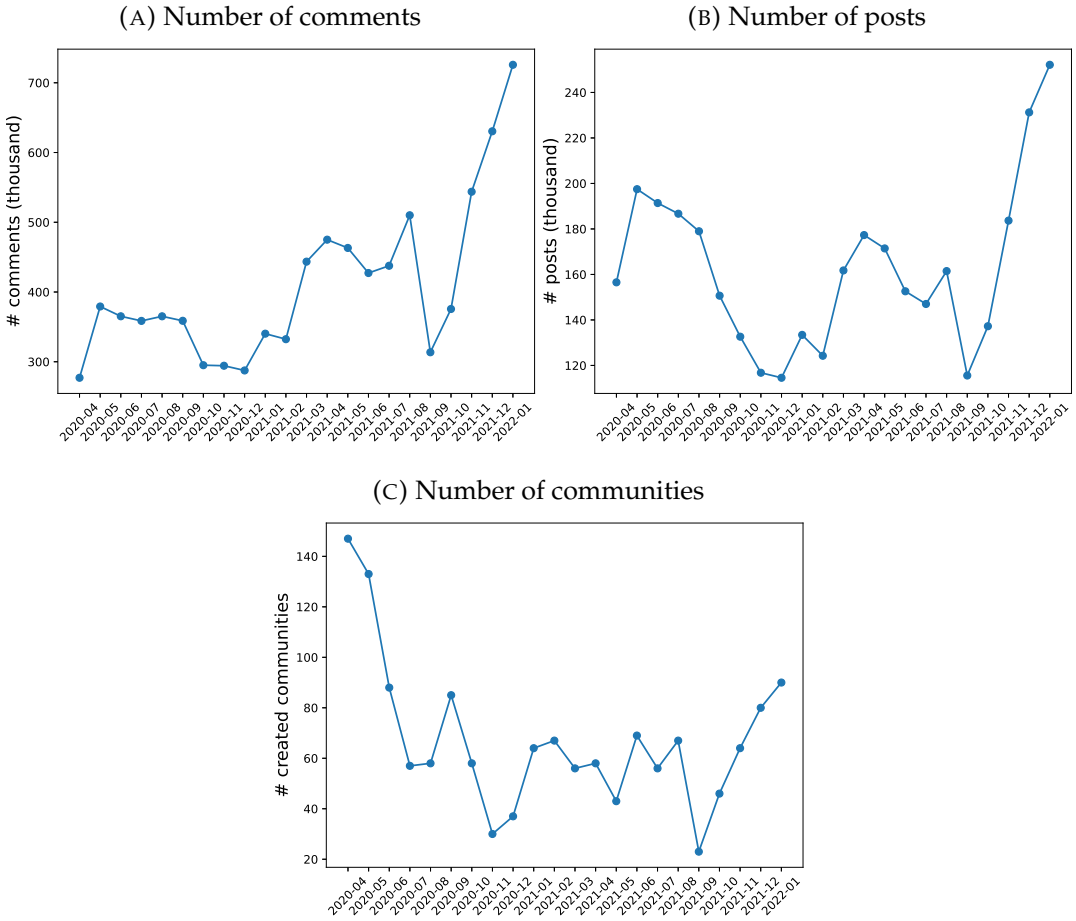


FIGURE 3.5: The Hive dynamics from April 2020 to January 2022.

networks such as Facebook and Twitter. This pattern is also observed in the distribution of top active users by posts and comments in Figure 3.7.

The reward system is a unique feature of the Hive decentralized social network. Users could receive and claim their rewards in the Hive platform through the “Hive Backed Dollars (HDB)" tokens and the staked Hive tokens called “Hive Power." The former is related to user content, e.g., posts, comments, and the latter is paid directly to boost users’ popularity. While liquid Hive tokens could convert to HBD, Hive Power is only based on the number of Hive tokens users staked in the platform. Figure 3.8 displays the value of HBD and Hive Power used by each user account over time. The figure shows an interesting phenomenon: From April 2020 to June 2021, the value of HBD per account increases over time. However, from June 2021 to October 2021, this value decreases significantly, and such a sudden drop also happens with the value of Hive power per account. This is expected since the growth of active users in this period is increased (see Figure 3.1) due to the Hive cryptocurrency price surge during this time [170]. The figure also shows a complex pattern of average reward in the Hive network and suggests further studies to understand how reward can affect users’ behaviors in the decentralized social network. Such research can open a new direction to improve existing

	Total Count	Avg. new per days
# created users	766,080	1,246
# posts	3,574,862	5,813
# comments	8,999,321	14,633
# comment edits	259,191	421
# upvotes	162,242,767	263,809
# downvotes	1,182,265	1,922
# communities	2,527	4

TABLE 3.2: Hive social network statistics until January 31, 2022.

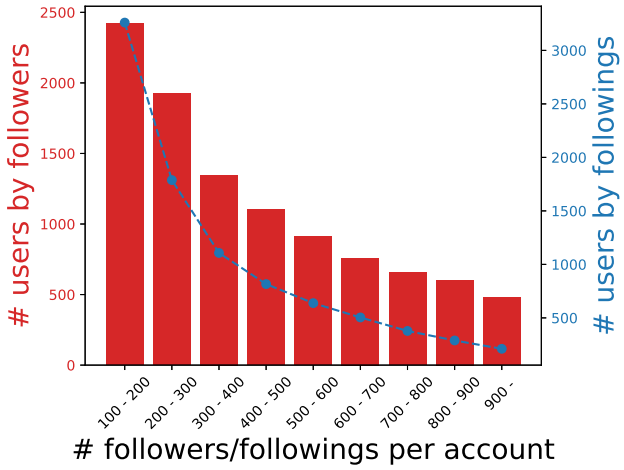


FIGURE 3.6: Number of Hive users based on the number of followers/followings per account.

centralized social networks.

Social Network Analysis

One of the outstanding advantages of a blockchain-based social network is information transparency. For instance, building networks of subscribers with the communities or authors with readers or followers with followings can be exploited instantly from the public data of the blockchain. In contrast, in traditional social networks, researchers usually require the approval of the organizations or firms that own such information.

Figure 3.9 illustrates a directed network with 527 nodes and 933 links of various active communities with their new subscribers on the Hive blockchain social network on May 2, 2021. A more intensified red color represents the Pagerank influence score of each node. Accordingly, the most influential node in the network is *#hive-168042*, a community named Planetauto that is specialized in providing automotive content such as car guides, car reviews, events, and games for cars. Similarly, Table 3.3 shows the top-three influential communities every month from January to December in 2021 using the networks sharing approximate graph structures in Figure 3.9 but more extensive and complex. Likewise, there are three most influential communities in the

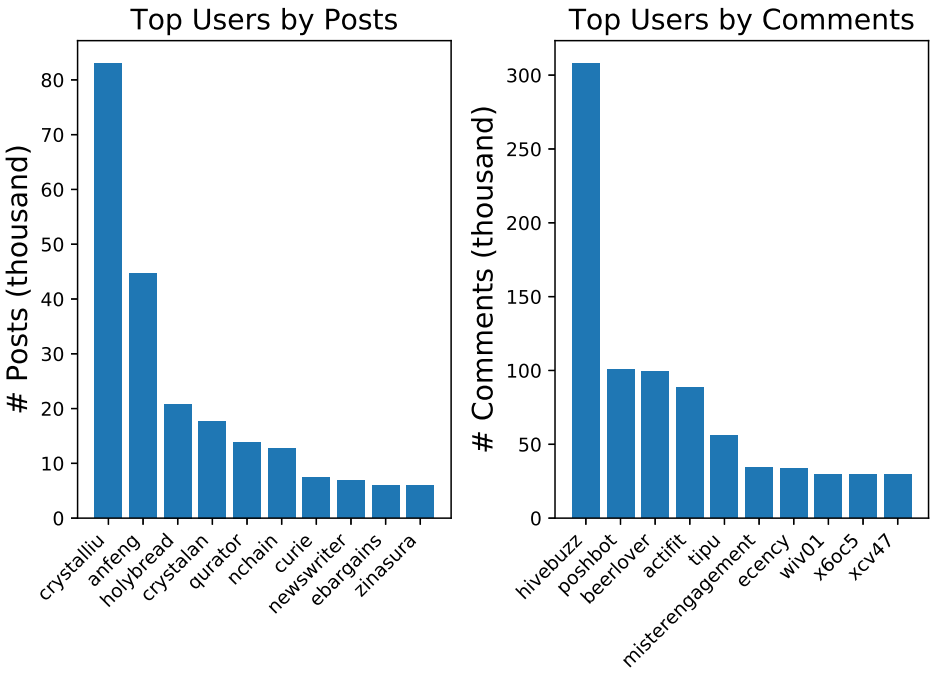


FIGURE 3.7: Top 10 users by posts and comments.

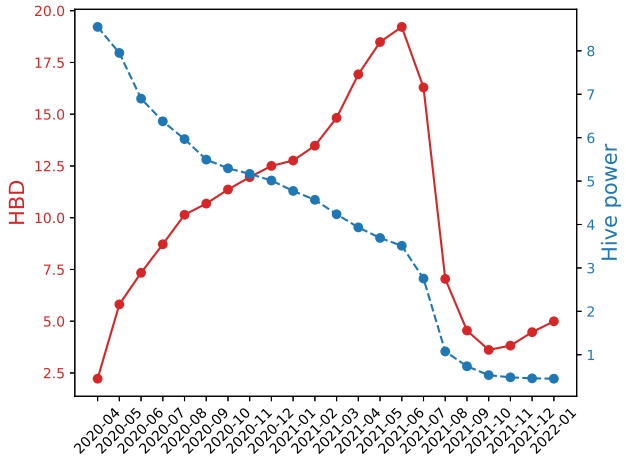


FIGURE 3.8: Average reward claimed per account in Hive social network from April 2020 to January 2022.

Hive decentralized network in this period, including *LeoFinance(#hive-167922)*, one of the largest crypto and finance content communities, *GEMS(#hive-148441)*, a community with a wide range of topics from lifestyle, cooking, and food hobby to history and philosophy in many different languages, and *Splinterlands(#hive-13323)*, a community specialized in Splinterlands being a digital collectible card game based on Hive blockchain. Interestingly, *Aquatic Sentinels (#hive-154473)*, a new and only-26-subscribers community specialized in sharing the beauty, diversity, and science of the aquatic and marine ecosystems, reached third place in May 2021. The reason originates from

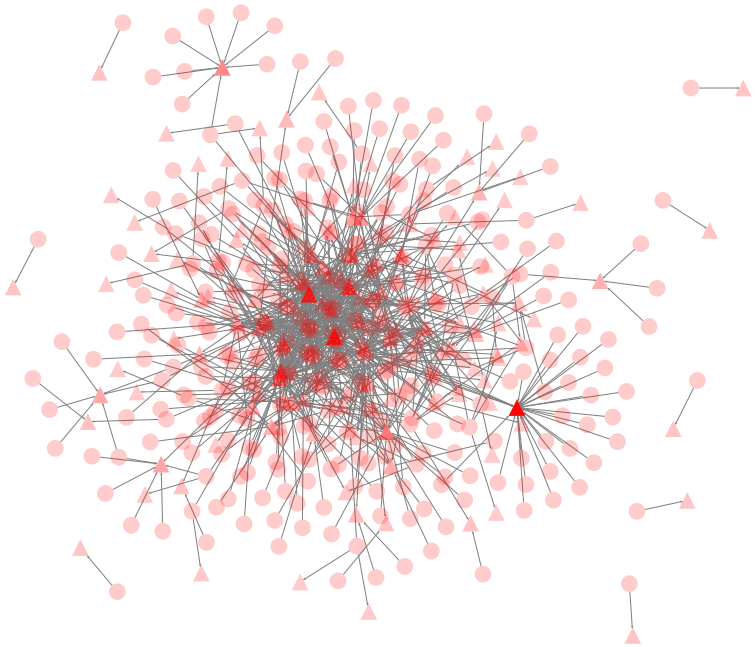


FIGURE 3.9: Network of active communities on Hive blockchain and their new subscribers on May 2, 2021. The circle nodes represent the subscribers, and the triangle nodes represent the communities. The intensity of the red color illustrates the influence of the node in the network.

another environment-related famous community with over 3,500 users called *Amazing Nature(#hive-127788)*, which has subscribed to the *Aquatic Sentinels* community at this time.

Comparison with Available Hive Statistics Analysis

There are minor differences between the statistics reported in Figures 3.1 and 3.5 and the daily and weekly reports of two users *arcange* and *penguinpablo* in *PeakD* [173] and *Hive.blog* [174]. However, we present monthly statistics in this chapter, while the existing statistics reports include daily and weekly statistics. This may cause a data mismatch misconception, but our investigations do not indicate significant differences.

3.4.3 Splinterlands - A Hive-based decentralized card game

Splinterlands is a collectible card game that leverages the power of Hive blockchain in storing all of its information. This is the principal technical difference of Splinterlands compared to the other blockchain-based games. For example, Gods Unchained, another collectible card game built upon Ethereum, stores only the players' assets and purchases information in blockchain while the rest, e.g., the battle result, is stored on the game publisher servers. Table 3.4 shows Splinterlands's overall statistics, which clearly indicates its popularity.

Gameplay: After signing up for the game, each new player can select numerous cards to start battles. Each card's properties fall into four categories:

Networks (By Month)	Network Statistics		Top Influential Communities		
			Rank	*Hive ID*	*Name*
January	*# Nodes*	3590	**1st**	**hive-180164**	**Hive Book Club**
	# Edges	10461	2nd	hive-167922	LeoFinance
	Avg. Degree	2.9139	3rd	hive-196037	DTube
February	*# Nodes*	3690	**1st**	**hive-167922**	**LeoFinance**
	# Edges	11220	2nd	hive-196037	DTube
	Avg. Degree	3.0407	3rd	hive-148441	GEMS
March	*# Nodes*	4993	**1st**	**hive-167922**	**LeoFinance**
	# Edges	18560	2nd	hive-145666	Photo Lovers
	Avg. Degree	3.7172	3rd	hive-148441	GEMS
April	*# Nodes*	6329	**1st**	**hive-148441**	**GEMS**
	# Edges	27134	2nd	hive-174578	OCD
	Avg. Degree	4.2872	3rd	hive-167922	LeoFinance
May	*# Nodes*	6065	**1st**	**hive-148441**	**GEMS**
	# Edges	25535	2nd	hive-174578	OCD
	Avg. Degree	4.2102	3rd	hive-154473	Aquatic Sentinels
June	*# Nodes*	4775	**1st**	**hive-130560**	**Hive Diy**
	# Edges	16724	2nd	hive-131619	Blockchain Gaming
	Avg. Degree	3.5024	3rd	hive-148441	GEMS
July	*# Nodes*	4919	**1st**	**hive-110011**	**Aliento**
	# Edges	18330	2nd	hive-148441	GEMS
	Avg. Degree	3.7264	3rd	hive-174578	OCD
August	*# Nodes*	5521	**1st**	**hive-13323**	**Splinterlands**
	# Edges	20400	2nd	hive-148441	GEMS
	Avg. Degree	3.6950	3rd	hive-104151	Beyond Horizon
September	*# Nodes*	4171	**1st**	**hive-13323**	**Splinterlands**
	# Edges	12964	2nd	hive-148441	GEMS
	Avg. Degree	3.1081	3rd	hive-174578	OCD
October	*# Nodes*	4773	**1st**	**hive-181450**	**Education & Training**
	# Edges	15176	2nd	hive-13323	Splinterlands
	Avg. Degree	3.1796	3rd	hive-184127	Regional Press
November	*# Nodes*	6951	**1st**	**hive-13323**	**Splinterlands**
	# Edges	23605	2nd	hive-167922	LeoFinance
	Avg. Degree	3.3959	3rd	hive-148441	GEMS
December	*# Nodes*	8776	**1st**	**hive-181450**	**Education & Training**
	# Edges	31594	2nd	hive-184127	Regional Press
	Avg. Degree	3.6000	3rd	hive-173286	Gods On Chain

TABLE 3.3: Top influential communities based on networks of active communities and their new subscribers on Hive blockchain in 2021.

(i) Rarities determine how rare the card is. There are four levels of rarities: common, rare, epic, and legendary. (ii) Each card has seven stats: Mana cost, Speed, Armor, Health and Attach including Melee, Ranged, and Magic. (iii) Fire, Earth, Water, Life, Death, Dragon, and Neutral define the faction of this card. (iv) Each card has more than 46 abilities to increase the fun and randomness of battles. Before a battle between two players, each player is provided with a fixed amount of mana, and each player chooses the same number of cards to organize on the battlefield. The result of the battle is determined by position, strength, cards' ability, and some randomness injected by the system. There are three types of battle: ranked, practiced, and friendly matches. The last two do not affect the ranking of players. The players' cards can be traded with other ones.

Category	Feature	Count
Overview Statistic	# of active users	380,476
	# of daily games	9,240,084
	# of cards	283+
Account-related Actions	# Claim Reward	58,248,195
	# Upgrade Account	664,180
	# Add Wallet	517,058
Battle Actions	# Match Finding	186,016,801
	# Match Starting	2,477,844
	# Surrender	1,685,134
Asset Actions	# Burn Cards	323,701
	# Lock Assets	137,777
Purchase Actions	# Sell Cards	6,387,080
	# Purchase Record	52,786

TABLE 3.4: Splinterlands statistics until January 31, 2022.

Data Analysis: Each blockchain transaction records an action that happened in the game. We can cluster these actions into the following four categories. Table 3.4 represents some notable activities for each category:

1. *Account-related actions*: Some example operators in this category are rewarded with more than 36 million transactions, while upgrade account operator has more than 626K transactions on the Hive blockchain, and adding wallet is nearly 480K operators.

2. *Battle actions*: It contains activities related to a battle, e.g., players of a battle, each player's order of card deck in the match, the battle result, and the battle type. Some examples include more than 178 million match-finding transactions, where users have played nearly 2.5 million matches.

3. *Asset actions*: It is the card information that each player has. So, for example, players can destroy cards, which they do not want to use. According to Hive transactions, there are 314 thousand actions recorded on the system. Lock asset is also done more than 92K times.

4. *Purchase actions*: Since the game allows its players to buy, sell, or transfer their assets, all transactions are stored in the blockchain to avoid manipulation, even from the game publisher side. As stated in Table 3.4, around 4.5 million transactions are related to the users' card selling activities. It expresses the users' high trading activities in this game.

3.4.4 NFTShowroom

NFTShowroom is a marketplace for artists selling their digital art. The platform associates each digital art with a unique Non-Fungible Token (NFT). Since Hive blockchain is not designed as a decentralized computational system as Ethereum or EOS.IO, a smart contracts side-chain layer called Hive

Engine is used to issue token SWAP.HIVE of NFTShowroom. Then, the ownership of the artwork can be verified employing the Hive Engine smart contracts, and the primary layer of the Hive blockchain is leveraged for the verification in the case of NFTShowroom. Moreover, the purchase history of the digital art is trackable via the transactions created on the Hive Engine nodes before automatically sending them to the Hive main chain. When digital art is transferred, sold, or bought, the transactions are recorded in the Hive blockchain. This information is publicly transparent and could be verified by other users.

Till end of January 2022, NFTShowroom consists of more than 11,866 artworks sold. On average, ten artworks are purchased through the system every day. Based on our dataset, the total number of NFTShowroom tokens minted is 45,961, and such number is increasing over time because of its rapid development.

Chapter 4

Link Prediction for Social Network Analysis in Criminal Investigation

The identification of potential offenders, who are more likely to form a new group and co-offend in a crime, plays an essential role in narrowing down law enforcement investigations and improving predictive policing. Once a crime is committed, focusing on linking it to previously reported crimes and reducing the inspections based on shreds of evidence and the behavior of offenders can also greatly help law enforcement agencies. However, classical investigative techniques are generally case-specific and rely mainly on police officers manually combining information from different sources. Therefore, automatic methods designed to support co-offender research and crime linkage would be beneficial. Building on the foundation of social network analysis as developed during the development of SoChainDB presented in Chapter 3, in this chapter, we propose two graph-based machine learning frameworks to address these issues based on a burglary use case, the first being *transductive link prediction*, which seeks to predict emergent links between existing graph nodes (which represent criminals or criminal cases), and the other being *inductive link prediction*, where connections are found between a new case and existing nodes.

4.1 Introduction

Various theoretical and empirical studies indicate that most crimes concentrate in time and space according to the so-called “law of crime concentration" [179]. Furthermore, crime commission and victimization are concentrated among a minority of a population [180, 181]. The evidence of the concentration of crime and offenders paved the way for developing increasingly sophisticated crime prediction approaches.

Predictive policing aims to predict the risk of future crime incidents based on past crimes and other information [182]. It supports the identification and prioritization of potential targets for crime investigation through the application of advanced analytical techniques [183]. Traditionally, police officers have used simple, manual approaches to identify hotspots (i.e., areas with a higher likelihood of crime), for instance, by pinning incidents on maps [184, 185]. These approaches have been improved through the application of a variety of quantitative techniques [186] that can handle the crime prediction problem more efficiently, including various classic machine learning methods such as Random Forests [187], Naive Bayes [188] and Support Vector Machines (SVMs) [189]. As a result of these advances, predictive policing has found several applications in law enforcement to predict the time and location of future crimes [190] while generating a lively debate about its effectiveness and potential biases [191–193]. Despite the increasing attention to predictive policing, research has rarely addressed the possibility of identifying potential offenders and, more specifically, co-offending in crime, that is, when two or more individuals participate in the same crime [182]. Yet, similar issues are central to another stream of research that focuses on crime linkage, a

process of associating two or more crimes based on evidence and offender behavior [194–196]. However, this approach has largely relied on case-specific qualitative information (e.g., identifying the modus operandi of a specific offender and linking it to criminal events) and has barely evolved to more general methods [93].

As a crime generally involves an offender and a target and often occurs in a certain place and time, predictive policing techniques should answer, at least in part, one or more of the following questions: (1) who will commit a crime, (2) who will be the victim, (3) what type of crime will be committed, (4) in what location and (5) at what time will a new crime take place [197]. While most of the previous applications of predictive policing have focused on the last three questions and most crime linkage approaches have addressed the first question through case-specific methods, in this chapter, we concentrate mainly on answering the first question about who will offend and with whom through advanced machine learning. In particular, the present research aims to address the following research questions:

> **RQ1:** Knowing a network of offenders and their previous collaborations, can we predict potential future burglary attempts made by existing offenders in the network?
>
> **RQ2:** Knowing the historical information of crimes and their offenders, can we narrow down the inspections of a new case to a list of potential offenders?

To this end, a comprehensive burglary dataset of more than 30,000 real case reports is processed and further transformed into a bipartite graph of offenders and criminal cases to build a network of criminals based on the collected information, which is then used to determine the co-offense likelihood for known criminals. By proposing different machine learning methods, our goal is to contribute to predictive policing and crime linkage research with a general, parsimonious, and automated approach. In doing so, we first propose an unsupervised link prediction framework that uses node neighborhoods and path information to identify possible links based solely on the network topology. On the other hand, several studies show that if labeled instances are available, supervised link prediction approaches outperform unsupervised methods [198, 199]. Although some studies have applied supervised learning for link prediction (e.g., [199, 200]), the use of these methods for co-offense prediction is still scarce [182]. Moreover, the success of the most used supervised algorithms usually depends on the data preparation and feature engineering method that properly describes the phenomenon. Therefore, we attempt to overcome these limitations by exploring more advanced analytical methods, specifically deep neural networks. Since we have prior knowledge of the offending history of criminals, we call this approach *transductive link prediction*. Furthermore, we propose an *inductive link prediction* framework to assess whether offenders of a newly placed crime can be predicted based on the textual reports of the crimes presented as node attributes.

4.2 Data Collection and Network Creation

4.2.1 Burglary Dataset

The dataset used in this study, provided by the Israel National Police and duly anonymized according to the standards of Israeli national law and GDPR and further approved by a legal advisor of the Israel National Police, contained around 30,000 reports of solved burglary cases that occurred in Israel between 2012 and 2021. The information contained in the reports included a crime identifier, the respective anonymized identifiers of the offenders, the min-max-scaled site coordinates of the crime that prevent precise retrieval of the localization of the site, timestamps, and a parameterized free-text description of the case in the form of an embedding vector, in particular, a SIF embedding [201]) described next. Even though we used a burglary database similar to Solomon *et al.* [93], our study employed a completely different link prediction approach as they looked at the link prediction task from a classical machine learning perspective, while our work is framed based on graphs, where crimes and offenders are nodes represented by features[1], hence explicitly considering the structure of the criminal network and not only the features of the sample.

Text Embedding: The properties of the crime nodes in our graphs are transformed into embedding vectors built from the textual description of the incident. Even though the spatiotemporal information of a crime is very important for link detection and prediction, we have ignored these features, which could be seen as a routine independent investigative filtering procedure. Similar to Solomon *et al.* [93], we parameterized the textual information of the crime using Smooth Inverse Frequency (SIF) [201] [2]. The SIF method can encode sequences of words in a sentence or paragraph into a single vector, mathematically representing the crime description. In short, this method intelligently combines the embeddings of each word within a sequence to identify the most relevant ones, a simple semantic text similarity task that has proven to work well [203].

4.2.2 Generated Networks

Three different networks were generated based on the original burglary data and text embeddings, including:

[1]While this study uses only offender or crime descriptions, future research could also incorporate criminal characteristics.

[2]More sophisticated alternatives to text embedding, such as those based on Bidirectional Encoder Representations from Transformers (BERT) [202], have recently been proposed. However, since the optimization of text embeddings as node attributes is not the main goal of this article, we followed the suggestion of Solomon *et al.* to use SIF instead of BERT. Their findings could be due to the fact that the pretrained AlephBert did not fit the current corpus, while SIF relied on a domain-specific word2vec model trained on a large corpus of our criminal texts. Since police reports lacked the proper language formalism, we believe that the benefits of BERT are diminished, especially when it was not fine-tuned to our data. Further elaboration on text embedding optimization is suggested for future work.

1. A *crime-offender network* (Figure 4.1) with 41,324 nodes and 34,156 edges, where nodes represent both crimes and offenders, and links indicate whether an offender participated in a crime. This network is a bipartite graph with two different types of nodes later divided into two networks for further analysis.

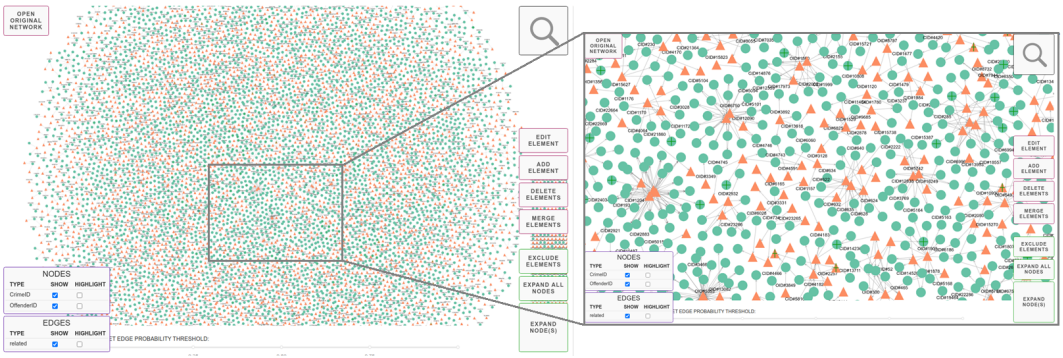


FIGURE 4.1: The crime-offender bipartite network[3]. Green circles and orange triangles represent crime cases and offenders, respectively. Edges indicate when an offender participated in a crime.

2. An *offender network* (Figure 4.2), where the nodes represent the offenders and the links indicate whether two offenders are involved in one or more burglary cases. To generate the offender graph, we used the original undirected crime-offender network to connect two offender nodes each time they shared a crime and then filtered the crime nodes. The offender network resulted in a total of 17,232 nodes and 21,302 edges. We used this dataset in the link prediction experiments as the network is large enough for this purpose. Furthermore, we specifically conducted the unsupervised transductive link prediction experiments on this dataset as we only accessed the network structure and not additional embedded attributes for offenders.

3. A *crime network* (Figure 4.3) with nodes representing offenses and edges indicating the number of shared offenders between the crimes. This network included 23,380 nodes and 42,604 edges. It was used for link prediction experiments, especially inductive link prediction, since nodes, i.e., crime cases, consisted of embedded descriptions as attributes.

4.3 Link Prediction Methods

We propose two approaches for link prediction: *transductive link prediction* (Section 4.3.1), which considers prior knowledge of the offender network to predict emerging links between existing nodes (i.e., offenders), and *inductive link prediction* (Section 4.3.2), where the links of a new crime to existing crimes can be predicted based on available meta-information (i.e, node attributes).

[3]Snapshots are taken using our open-source network analysis tool, which can be downloaded from https://github.com/erichoang/criminal-network-visualization.

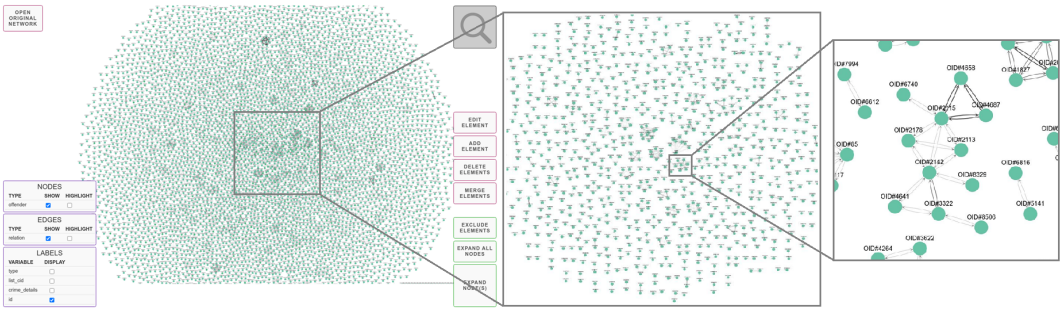


FIGURE 4.2: The offender network, in which nodes represent offenders, and edges indicate if two offenders share a crime.

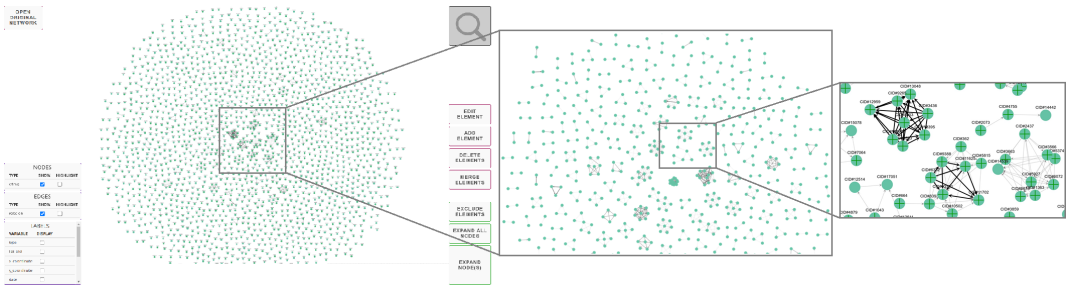


FIGURE 4.3: The crime network. The nodes represent placed crimes and the edges indicate whether two crimes share an offender.

4.3.1 Transductive Link Prediction

As mentioned in Section 2.1.1, in principle, all link prediction methods work by assigning a $score(u, v)$ to pairs of nodes (u, v) and then choosing the top-scored pairs as predicted links for u [35]. The score value is computed based on the input network and often measures the proximity or similarity between u and v. Different measures of similarity have been proposed, the most relevant in the literature and applicable for crime prediction being the following:

- **Jaccard similarity** measures the probability that both u and v have past connections to individual f. For this, an individual f is randomly selected from a set of all individuals with past connections to either u or v. The probability is then calculated as the ratio of the number of individuals that both u and v have observed interactions with and the number of individuals with whom either u or v have observed interactions. In particular:

$$Jaccard(u, v) = \frac{|\Gamma(u) \cap \Gamma(v)|}{|\Gamma(u) \cup \Gamma(v)|}, \tag{4.1}$$

  where Γ indicates the neighborhood function of a node.

- **Adamic Adar similarity** [36] is a variant of the Jaccard ratio, where similarity is determined as a weighted count of the individuals with whom both u and v have had interactions. Each individual is weighted

by the inverted number of all other observed individuals:

$$Adamic(u,v) = \Sigma_{z \in \Gamma(u) \cap \Gamma(v)} \frac{1}{\log |\Gamma(z)|}. \tag{4.2}$$

In this setting, a popular individual with many connections other than to u and v contributes less to the similarity ratio than an individual f who only has past connections to u and v. It is reasonable since, in this case, f could probably be the driver of the relationship between u and v.

- **Preferential attachment similarity** [37] follows what is called the Rich-Get-Richer phenomenon. This means that the probability that u will connect with v in the future is proportional to the popularity of these two individuals. In its simplest mathematical form, this measure is determined by the product of the number of other individuals with whom u has had interactions and the same number with v:

$$Preferential(u,v) = |\Gamma(u)||\Gamma(v)|. \tag{4.3}$$

- **Resource allocation similarity** [38] aims to measure resources flowing from u to v through other people with whom u and v have connections. It has a very similar mathematical formulation to the Adamic Adar similarity, except that the number of other individuals is now weighted by their individual interactions:

$$Resource\_Allocation(u,v) = \Sigma_{z \in \Gamma(u) \cap \Gamma(v)} \frac{1}{|\Gamma(z)|}. \tag{4.4}$$

- **Soundarajan-Hopcroft similarity** [39] is an improved variant of Adamic Adar similarity that considers information about the community structure of the input network. The rationale is that the more community members the two individuals have in common, the more likely they are to form some sort of connection. Mathematically, the score is determined by the number of individuals both u and v have interacted with, plus the number of communities they both belong to:

$$Soundarajan\_Hopcroft(u,v) = \Sigma_{z \in \Gamma(u) \cap \Gamma(v)} \frac{f(z)}{|\Gamma(z)|}, \tag{4.5}$$

where $f(z)$ equals 1 if z belongs to the same community as u and v.

These measures are based on the connections between individuals in the network, so they can be improved by considering the attributes and behavior of individuals when calculating the similarity values. Several works have leveraged this approach. The tensor factorization method, for instance, has gone hand in hand with a high computational cost considering its technical complexity, which is why we have discarded it in this study.

Prediction by Transductive Algorithm

Our approach for similarity-based transductive link prediction is simple yet effective. For each pair of nodes, (v, u), in the node set V, we first calculate their similarity $\text{sim}(v, u)$ using one of the following similarity metrics: the Jaccard coefficient, Adamic-Adar index, resource allocation index, preferential attachment or the Soundarajan-Hopcroft coefficient. Then, for each node, the top k similar nodes regarding the computed similarity metric are selected, which indicates the target nodes of the outgoing edges of that particular node. The pseudo-code of our transductive approach is presented in Algorithm 1.

Algorithm 1 Transductive Link Prediction Algorithm

Require: $G = (\mathcal{V}, \mathcal{E})$, k, sim ▷ sim represents the similarity metric function
  Calculate $\text{sim}(u, v), \forall u, v \in \mathcal{V}$
  pred $\leftarrow$ []
  for $v \in \mathcal{V}$ **do**
    for $u \in$ select-top$(\text{sim}(v, \cdot), k)$ **do** ▷ select the top k similar nodes
      pred $\leftarrow$ pred $+ (v, u)$
    end for
  end for
  Return pred

4.3.2 Inductive Link Prediction

In attributed graphs, both the network structure and attribute information can be used for link prediction. An attributed graph is represented by $G = (\mathcal{V}, \mathcal{E}, \mathcal{X})$, where $\mathcal{V} = \{v_1, ..., v_n\}$ represents the node set, $n = |\mathcal{V}|$, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ the edge set, $\mathcal{X} = \{x_1, ..., x_n\}$ the attribute feature matrix, $x_i \in \mathbb{R}^m$ the attribute feature vector of node v_i and m the number of attributes in the graph. Given a graph G and a pair of nodes (v_i, v_j), the goal of link prediction is to estimate the likelihood of a link exists between v_i and v_j. Most current methods focus on transductive link prediction, where both nodes v_i and v_j are already known. However, in many real-life situations, inductive prediction is also required considering new nodes, where attribute information is available, but one or both of the nodes v_i and v_j have not been observed, at least during the training process.

To overcome this limitation, we implement an attributed graph embedding method called Dual-Encoder graph embedding with ALignment (DEAL) [204], which can be used for both inductive and transductive link prediction. This framework embeds the graph of existing nodes in the vector space and extracts its structure information (i.e., structure-oriented node embedding). It then computes an embedding vector for new query nodes where the only information available is their attributes (i.e., attribute-oriented node embedding), which is ultimately compared to the previously computed structure embedding. In particular, the DEAL approach involves three main components, including two types of node embedding encoders and one alignment

mechanism. The first encoder aims to output the attribute-oriented node embedding ($\mathcal{H}_a$), while the second relates to the structure-oriented node embedding ($\mathcal{H}_s$): $\mathcal{H}_a$ computes the embedding vectors with attributes of the new nodes and $\mathcal{H}_s$ computes the node embedding vector that preserves the structure information. Finally, the alignment mechanism is employed to contrast the two embedding vectors and build the connections between the attributes and the links. Both encoders are updated during training, so the generated embeddings are aligned.

Attribute-Oriented Encoder

The attribute-oriented encoder $\mathcal{H}_a$ ingests an attribute vector x_i from node v_i and generates a node embedding $z_i^a = \mathcal{H}_a(x_i)$. We used an MLP with a nonlinear activation layer to learn $\mathcal{H}_a$:

$$\mathcal{H}_a(x_i) = \sigma(\mathbf{W}_a^2(\sigma(\mathbf{W}_a^1 x_i + b_a^1)) + b_a^2), \tag{4.6}$$

where W_a^1, W_a^2, b_a^1 and b_a^2 are hyperparameters of the model and $\sigma(.)$ is the exponential linear unit. We opt for a simple MLP approach instead of complex neural networks since well-known Graph Neural Networks (GNN) models such as Graph Convolutional Networks (GCN) [8] and Graph Attention Networks (GAT) [40] suffer from scalability limitations (see Section 4.4.2).

Structure-Oriented Encoder

The structure-oriented encoder $\mathcal{H}_s$ generates node embeddings that only preserve the structural information of the graph G without regard to the attributes of the nodes. We use the one-hot encoding of the nodes $\mathcal{I}_\mathcal{V} = \{I_1, ..., I_n\}$ as the input of the encoder and further map node v_i to its node embedding vector $z_s^i = \mathcal{H}_s(I_i)$. We then employ a linear model to compute the encoder $\mathcal{H}_s$:

$$\mathcal{H}_s(I_i) = g(W_s) I_i, \tag{4.7}$$

where $g(.)$ is used to re-parameterize W_s and accelerate the convergence of stochastic gradient descent optimization.

Alignment Mechanism

We align the embeddings of the two types of encoders with learning the connections between the node attributes and the graph structure. In doing so, we apply a ranking-motivated loss function that learns the graph embedding through ranking, which can help capture the relationships between each pair of nodes in the training samples. Inspired by the contrastive loss [205] that maps similar input samples to nearby points in the output vector space and dissimilar samples to distant points, we map the linked graph nodes to close points in the output vector space and the unlinked nodes to points far apart. However, it should be considered that the unlinked nodes (negative pair-wise samples) have different distances in the graph. Consequently, we

employ the following loss function for a given mini-batch of node pairs $B = \{(v_{p_1}, v_{q_1}), ..., (v_{p_k}, v_{q_k})\}$, where $p_i \neq q_i$ and $i \in [1, k]$:

$$\mathcal{L}_B(\mathcal{Z}) = \frac{1}{|B|}\Sigma_{(v_{p_i}, v_{q_i}) \in B}[(1 - y_i)\alpha(v_{p_i}, v_{q_i})\phi_1(-s(z^{p_i}, z^{q_i})) + y_i\phi_2(s(z^{p_i}, z^{q_i}))], \tag{4.8}$$

where $s(.,.)$ represents a similarity function that compares two node embeddings (z^{p_i} and z^{q_i}) and y_i the link relation label (i.e., $y_i = 1$ if two nodes are connected). In this thesis, we use the so-called cosine similarity function. Further, α represents a weight function to measure the importance of negative samples with different distances. We can define it as $\alpha(v_p, v_q) = \exp \frac{\beta}{d_{sp}(v_p, v_q)}$, where $d_{sp}(.)$ denotes the shortest path between two nodes and $\beta > 0$ is a hyperparameter. If two nodes are unreachable, then $d_{sp}(v_p, v_q) = \infty$. Both ϕ_1 and ϕ_2 are derived from function $\phi(.)$, which considers different hyperparameters to link regularization. We then use the generalized logistic loss function ($\phi(x)$, with $\gamma > 0$ and b as loss margin parameters) to tune regularization:

$$\phi(x) = \frac{1}{\gamma}\log(1 + \exp^{-\gamma x + b}). \tag{4.9}$$

Instead of optimizing $\mathcal{L}_B(\mathcal{Z}_s)$ and $\mathcal{L}_B(\mathcal{Z}_a)$ independently, we design a joint alignment method that aims to maximize the similarity between $z_s^{p_i}$ and $z_a^{q_i}$ of two linked nodes v_{p_i} and v_{q_i}:

$$\mathcal{L}(\mathcal{Z}_s, \mathcal{Z}_a) = \frac{1}{|B|}\Sigma_{(v_{p_i}, v_{q_i}) \in B}[(1 - y_i)\alpha(v_{p_i}, v_{q_i})\phi_1(-s(z_s^{p_i}, z_a^{q_i})) + y_i\phi_2(s(z_s^{p_i}, z_a^{q_i}))]. \tag{4.10}$$

The overall objective of our inductive model follows:

$$\mathcal{L} = \theta_1\mathcal{L}_B(\mathcal{Z}_s) + \theta_2\mathcal{L}_B(\mathcal{Z}_a) + \theta_3\mathcal{L}_{align}(\mathcal{Z}_s, \mathcal{Z}_a), \tag{4.11}$$

where $\theta_1, \theta_2, \theta_3$ are hyperparameters to parameterize the weights of different losses further.

Algorithm 2 Inductive Link Prediction Algorithm (DEAL)

Require: Graph $G = (\mathcal{V}, \mathcal{E}, \mathcal{X})$, a set of mini-batches B, loss weight θ
  for each batch in B **do**
    $\mathcal{Z}_s \leftarrow \mathcal{H}_s(\mathcal{I}_\mathcal{V})$
    $\mathcal{Z}_a \leftarrow \mathcal{H}_a(\mathcal{X})$
    $\mathcal{L} \leftarrow \theta.[\mathcal{L}_B(\mathcal{Z}_s), \mathcal{L}_B(\mathcal{Z}_a), \mathcal{L}_{align}(\mathcal{Z}_s, \mathcal{Z}_a)]$
    Update $\mathcal{H}_s$ and $\mathcal{H}_a$ with stochastic gradient $\nabla\mathcal{L}$
  end for
  Update $\mathcal{Z}_s$ and $\mathcal{Z}_a$ via $\mathcal{H}_s$ and $\mathcal{H}_a$
  Calculate score$(u, v), \forall u, v \in \mathcal{V}$ via Equation 4.12

Link Prediction

To determine if there is a link between two nodes v_p and v_q exists, we calculate the following score:

$$score(v_p, v_q) = \lambda_1 s(z_s^p, z_s^q) + \lambda_2 s(z_a^p, z_a^q) + \lambda_3 s(z_s^p, z_a^q), \tag{4.12}$$

where $\lambda_1, \lambda_2, \lambda_3$ are hyperparameters of the similarity score. In inductive link prediction, the link z_a^q to a new node v_q is calculated by setting $\lambda_1 = 0$. Algorithm 2 summarizes the steps in the DEAL framework. DEAL can also perform transductive link prediction by specifying λ_1 to a non-zero value.

4.4 Experimental Analysis

The performance of link prediction methods is often evaluated by their ability to retrieve links hidden on purpose. Accordingly, we first conceal a small portion of the links for a given individual and then use the prediction methods to identify potential hidden links. Section 4.4.1 and 4.4.2 discuss our findings on transductive[4] and inductive[5] link prediction experiments, respectively.

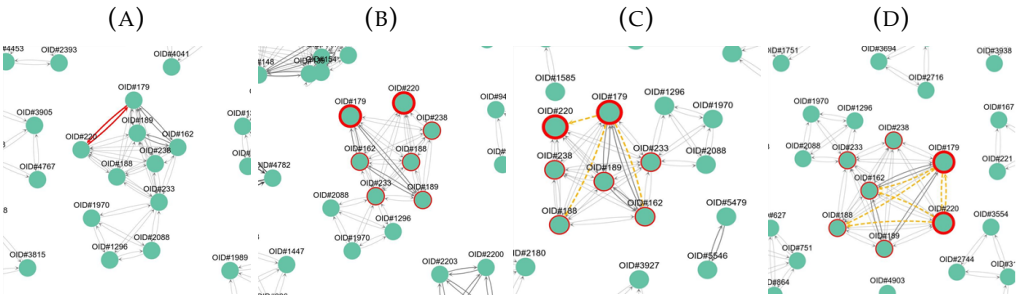


FIGURE 4.4: (a) Original offender network and two randomly selected nodes (*OID*#179 and *OID*#220), (b) Modified network by removing the links between the two selected nodes, (c) Transductive link prediction on *OID*#179 in a modified offender network, (d) Transductive link prediction on both *OID*#179 and *OID*#220 in the modified network. The link prediction method uses the Jaccard similarity measure, and the predicted links are the yellow-dash lines.

4.4.1 Transductive Link Prediction Results

Figure 4.4 shows an example of our transductive link prediction using the Jaccard similarity measure. Figure 4.4a shows the original network where two connected nodes, *OID*#179 and *OID*#220, are randomly selected. To evaluate our link prediction algorithm, we remove the existing links using the available options in our visualization platform. Then, we apply the transductive link prediction with Jaccard similarity sequentially on *OID*#179 and *OID*#220. The method returns the top k likely emerging edges (here, top three). Figure

[4] https://github.com/erichoang/criminal-network-visualization
[5] https://github.com/erichoang/criminal-link-prediction

4.4d shows that the method can correctly predict the links between the two nodes.

Link Removal	Top-k	Jaccard Similarity	Adamic Adar	Preferential Attachment	Resource Allocation	Soundarajan-Hopcroft
1%	1	**61.78 (1)**	61.03 (3)	28.69 (5)	60.84 (4)	61.12 (2)
	3	67.10 (4)	**67.38 (1)**	46.92 (5)	**67.38 (1)**	67.20 (3)
	5	68.32 (2)	68.32 (2)	58.32 (5)	68.32 (2)	**68.60 (1)**
5%	1	**58.69 (1)**	54.84 (3)	28.50 (5)	54.73 (4)	56.64 (2)
	3	65.87 (2)	65.52 (3)	47.1 (5)	65.50 (4)	**66.34 (1)**
	5	67.49 (4)	67.56 (2)	59.38 (5)	67.54 (3)	**68.03 (1)**
10%	1	**53.1 (1)**	48.50 (3)	27.67 (5)	48.37 (4)	50.23 (2)
	3	**62.30 (1)**	60.85 (3)	46.41 (5)	60.81 (4)	62.03 (2)
	5	64.11 (2)	63.70 (3)	57.87 (5)	63.66 (4)	**64.35 (1)**
15%	1	**48.36 (1)**	43.59 (3)	27.02 (5)	43.47 (4)	45.24 (2)
	3	**58.78 (1)**	57.27 (3)	45.54 (5)	57.18 (4)	58.20 (2)
	5	60.96 (2)	60.64 (3)	56.10 (5)	60.59 (4)	**61.02 (1)**

TABLE 4.1: Accuracy of transductive link prediction on five different similarity measures and their ranking. The best results are in bold.

Table 4.1 reports the results of the transductive link prediction on the offender network by randomly selecting a sample of links to be removed from the original network (1%, 5%, 10%, and 15%). The link prediction scores between the central nodes and their 2-hop neighbors (the so-called *candidate edges*) are computed for each removed link based on the respective similarity measure. For each setting (determined by the specific similarity measure and the percentage of links removed), accuracy will naturally increase if we search for the correct link among more potential links (i.e., larger k). Moreover, for a fixed similarity measure and a fixed k, we can see the accuracy degrades with more links removed as the network structure changes more and we have less knowledge between offenders. The only exception is the Preferential attachment measure when the removed links increase from 1% to 5%, where accuracy improves slightly. It can also be observed that Jaccard similarity achieves the best accuracy on top-1 predicted links for all cases (i.e., offender network with different ratios of removed links). Finally, the Soundarajan-Hopcroft similarity measure slightly improves the Jaccard similarity results on the top five predicted links in all test cases.

4.4.2 Inductive Link Prediction Results

In this experiment, we train the DEAL framework on the crime network. We first explain how data is split, and then we describe the comparison methods and their settings. Performance is measured by known metrics such as the Area Under the Receiver Operating Characteristic Curve (ROC-AUC) and Average Precision (AP) at the top-ranked predicted link level [35].

Data Splitting

We split the input burglary network into three subsets:

1. Training set: Burglary cases before 2019.01.01 (75% with the number of nodes being 17,531)
2. Validation set: Burglary cases from 2019.01.01 to 2020.01.01 (15% with 3,488 nodes)
3. Test set: Burglary cases after 2020.01.01 (10% with 2,361 nodes)

For the data processing and the creation of the subnetworks, we only consider the training set that does not include the nodes and edges of the validation and test sets. The validation set is used for fine-tuning the DEAL hyperparameters, while the test set is left for evaluation purposes. For all subsets, we consider the text embeddings of the burglary case summaries as node attributes. We run the experiments in two versions:

- **Training with transductive validation**, where both training and validation sets are combined in the training process. In other words, the validation set is a subset of the training set, but there are no links between its nodes and the nodes of the training set.
- **Training with inductive validation**, where the network used in the training process is completely independent and different from the validation set network.

Methods of Comparison

We implement several baselines and state-of-the-art methods to compare their performance against our method on the burglary dataset. For the inductive link prediction, besides the fundamental graph embeddings and graph neural networks as **DeepWalk** [5], **node2vec** [6], **LINE** [7], **GCN** [8], and **GAT** [40] presented in Section 2.1.2, the following comparison approaches are considered:

- **Radius Neighbours:** This simple implementation predicts which newly-added nodes will be connected to their nearest neighbor nodes within the ball in the node embedding space. The radius of the nearest-neighbor ball is defined using the embedding of the training data (i.e., the nodes already in the network) to maximize the prediction accuracy on the training dataset.
- **Linear Model:** We consider the link prediction as a classification task, where the classifier differentiates the node pairs connected by an edge from the pairs that are not. We first augment the input data by considering any subset of data from Section 4.4.2, selecting *all* node pairs (u, v) where an edge $e = (u, v)$ exists and concatenating their vector representations into a new vector. Such concatenations are mapped to the positive class, i.e., $y = 1$. Then, we *randomly* sample negative pairs from each subset, where there are no edges in between, and set their $y = 0$. Finally, we train different linear classification methods, such as

LASSO, Ridge, and SVM, on the augmented training set $< (u, v), y >$ and make inferences on the test set. Note that this approach does not have newly added node w, so only transductive experiments apply.

- **Graph2Gauss:** As mentioned in Section 2.2.2, the Graph2Gauss [95] method trains a Gaussian model over the training network and further expands it using the newly added nodes so that the dissimilarity with respect to the Gaussian model is minimized between all connected nodes.

Parameter Setting

For training our method as well as Graph2Gauss, we augment the training network into multiple triples (u, v, w), where u and v are connected while w is a "negative sample" drawn from those nodes that do not connect to neither u nor v. For validation purposes, we aim to predict whether there is a connection between each new-old pair of nodes from the test set. Further, we configure the parameters for the baseline models as follows:

- **Radius Neighbours:** The radius of the nearest-neighbors ball is 0.74, which is the optimal value that minimizes the prediction error on the training set.

- **Linear Models:** Three linear classification models are implemented, including LASSO, Ridge, and the linear SVM model. We adopt the sklearn [206] implementation of these three algorithms maintaining the default parameters. For Ridge and LASSO, we set $\alpha = 1$. For the SVM classifier, the regularization parameter is set as $C = 1$ and the radial basis function (RBF) as the kernel (see [207]).

- **Graph2Gauss:** In the Graph2Gauss method, the Gaussian model for a single node is built based on the node's embeddings and its k-hop neighbors. We follow the same parameter setting as in [95] and set the count of maximum hop $k = 2$. We implement a two-layer perceptron to represent each node using the mean and covariant of a Gaussian distribution in $L = 64$ dimensions. We then train the model for 500 epochs and use the Adam optimizer with a learning rate of 0.01 and no weight decay.

- **DeepWalk:** The embeddings are trained over 100 epochs with window size 10 and $K = 3$ negative samples. Overall, $n = 10$ random walks are simulated with walk lengths of $l = 80$.

- **node2vec:** We use the skip-gram language model with a window size of $win = 10$ and $K = 3$ negative samples over 100 epochs. $n = 10$ random walks of length $l = 40$ with return parameter of $p = 1$ and in-out parameter of $q = 4$ are simulated in order to build the corpora.

- **LINE:** The LINE model is trained for 50 epochs with a batch size of 1024. The Adam optimizer is used for gradient descent with a learning rate of 0.001 and no weight decay. The number of negative samples is set as $K = 3$, and second-order proximity is used.
- **Graph Convolutional Network (GCN)**: We use the authors' recommended settings in the GCN model. Specifically, the model is trained for 200 epochs and uses the Adam optimizer with a learning rate of 0.01, two hidden layers, a dropout of 0.5, and no weight decay. Since the burglary network is large, we utilized two decomposed layers [208] to solve memory issues in our GCN experiments.
- **Graph Attention Network (GAT)**: The GAT model, featuring eight attention heads, is trained for 200 epochs. Gradient descent is performed using the Adam optimizer, with a learning rate of 0.01, two hidden layers, a 0.6 dropout rate, and no weight decay.

Results Discussion

	Methods	Featured Parameters	ROC-AUC	AP
Conventional methods	Radius Neighbours	Maximum radius $r = 0.74$	-	0.5146
	LASSO	Regularization parameter $\alpha = 1$	-	0.5000
	Ridge	Regularization parameter $\alpha = 1$	-	0.6114
	SVC	Regularization parameter $C = 1$	-	0.6065
	Graph2Gauss	$d_{\text{output}} = 64, d_{\text{hidden}} = 64,$ $n_{\text{hidden layer}} = 2$	0.6711	0.6556
Combination of basic graph embeddings and case summary text embeddings	node2vec + SIF	Pooling function = avg	0.568	0.5631
	DeepWalk + SIF	Pooling function = avg	0.5837	0.5833
	LINE + SIF	Pooling function = avg	0.533	0.5333
	GCN + SIF	Learning rate = 0.01, $d_{\text{hidden}} = 64,$ $n_{\text{decomposed layers}} = 2$	0.6206	0.6207
	GAT + SIF	Learning rate = 0.01, $d_{\text{hidden}} = 64,$ $n_{\text{heads}} = 8$	0.6128	0.6076
DEAL framework variants	DEAL-tr (default setting)	$\theta = [0.1, 0.85, 0.05]$, $\lambda = [0.1, 0.85, 0.05]$	0.7580	0.7567
	(increase θ)	$\theta = [0.2, 1.7, 0.1]$, $\lambda = [0.1, 0.85, 0.05]$	0.7582	0.7569
		$\theta = [0.4, 3.4, 0.2]$, $\lambda = [0.1, 0.85, 0.05]$	**0.7583**	**0.7571**
	(change λ)	$\theta = [0.1, 0.85, 0.05]$, $\lambda = [0.2, 1.7, 0.1]$	0.7580	0.7567
	(change both)	$\theta = \lambda = [0.05, 0.425, 0.025]$	0.7576	0.7561
		$\theta = \lambda = [0.2, 1.7, 0.1]$	0.7582	0.7569
		$\theta = \lambda = [0.4, 3.4, 0.2]$	**0.7583**	**0.7571**
	DEAL-ind (default setting)	$\theta = [0.1, 0.85, 0.05]$, $\lambda = [0.1, 0.85, 0.05]$	0.7468	0.7477
	(increase θ)	$\theta = [0.2, 1.7, 0.1]$, $\lambda = [0.1, 0.85, 0.05]$	0.7470	0.7479
		$\theta = [0.4, 3.4, 0.2]$, $\lambda = [0.1, 0.85, 0.05]$	0.7470	**0.7480**
	(change λ)	$\theta = [0.1, 0.85, 0.05]$, $\lambda = [0.2, 1.7, 0.1]$	0.7468	0.7477
	(change both)	$\theta = \lambda = [0.05, 0.425, 0.025]$	0.7465	0.7473
		$\theta = \lambda = [0.2, 1.7, 0.1]$	0.7470	0.7479
		$\theta = \lambda = [0.4, 3.4, 0.2]$	0.7470	**0.7480**
		$\theta = \lambda = [0.8, 6.8, 0.4]$	**0.7471**	**0.7480**

TABLE 4.2: Inductive link prediction results on the crime network. The best results in the transductive and inductive settings are highlighted separately. Note that the ROC-AUC is not applicable for Radius Neighbour and linear approaches.

We compare the methods with respect to ROC-AUC and Average Precision (AP). However, simple baselines such as Radius Neighbours and Linear

Models can only deliver Average Precision. The results of our experiments are reported in Table 4.2, while our findings are as follows:

- LASSO and Radius Neighbours achieve the worst results in correctly predicting links in the crime network. This was expected since both use straightforward strategies in comparing node embeddings. However, more complex regression methods, such as Ridge Regression, which gives more importance to the correlation between variables, notably achieve better results. Moreover, an SVM classifier with a linear kernel (SVC) can achieve comparable results to Ridge Regression. Graph2Gauss reaches the best results among the conventional baselines.

- Low-dimensional node embedding methods, including LINE, node2vec, and DeepWalk, do not show promising results on the crime network. LINE achieves the worst results among all three tested node embedding methods, unlike the experiments and comparisons in the original paper [7]. LINE is supposed to be more scalable than DeepWalk. Still, since it is an adjacency-based node representation, it does not perform as well as random walk-based embeddings in smaller networks like criminal analysis networks. Furthermore, since these node embedding methods are unsupervised, a simple SVC or Ridge classifier can reach comparable results.

- DEAL achieves considerably better results than conventional linear models and node embedding methods. This indicates that the DEAL framework effectively combines the structure and attributes of the crime network and extracts their augmented relations better than the methods that only use structure or attributes. Therefore, aligning the network topology with the node attributes is an effective strategy.

- Using a transductive validation set marginally improves the DEAL results. The larger the training set, the greater the performance improvement.

- Finally, DEAL is not sensitive to its hyperparameters. Changing the parameters θ and λ does not have a noticeable impact on the performance of the DEAL link prediction framework in both its transductive and inductive tests.

Chapter 5

Multi-Level Heterogeneous Graph Embeddings for Fine-Grained Detection of Smart Contract Vulnerabilities

Learning heterogeneous graphs consisting of different types of nodes and edges enhances the results of homogeneous graph techniques. An interesting example of such graphs is control-flow graphs representing possible software code execution flows. As such graphs represent more semantic information of code, developing techniques and tools for such graphs can be highly beneficial for detecting vulnerabilities in software for its reliability. However, existing heterogeneous graph techniques are still insufficient in handling complex graphs where the number of different types of nodes and edges is large and variable. This chapter concentrates on the Ethereum smart contracts as a sample of software codes represented by *heterogeneous contract graphs* built upon both control-flow graphs and call graphs containing different types of nodes and links. We propose MANDO, a new heterogeneous graph representation to learn such heterogeneous contract graphs' structures. MANDO extracts customized metapaths, which compose relational connections between different types of nodes and their neighbors. Moreover, it develops a multi-metapath heterogeneous graph attention network to learn multi-level embeddings of different types of nodes and their metapaths in the heterogeneous contract graphs, which can capture the code semantics of smart contracts more accurately and facilitate both fine-grained line-level and coarse-grained contract-level vulnerability detection. Our extensive evaluation of large smart contract datasets shows that MANDO improves the vulnerability detection results of other techniques at the coarse-grained contract level.

5.1 Introduction

Graph learning has been an active research area for a long time. Learning *heterogeneous* graphs that consist of nodes and edges of different types has recently attracted extensive attention since such graphs contain richer information from the application domains than homogeneous graphs and, therefore, can achieve better learning results [11]. However, when it comes to complex heterogeneous graphs, where the graph structures have particular properties and the number of node types and edge types can be arbitrarily large and changing, it is still unclear if existing techniques can handle them well. Examples of such graphs can be found in control-flow graphs or call graphs representing possible software code execution flows and call relations. A control-flow graph depicts all possible sequences of statements or lines of code that might be traversed in one function during program executions. In contrast, a call graph represents every possible call relation among functions in a program.

This chapter aims to develop a new approach for learning such complex and dynamic heterogeneous graphs and apply them to address critical software quality assurance problems, such as detecting vulnerabilities in software

code that can be represented as control-flow graphs and call graphs. Expressly, we represent software code as a combination of heterogeneous graphs of multiple granularity levels that capture the control-flow and call relations in code. Then, we extract specially defined *metapaths* for such graphs that acquire relations between different types of nodes and their neighbors, and fuse various kinds of graph neural networks together to learn both of the node-level and graph-level embeddings. Further, we use the embeddings to train networks to recognize graphs or nodes that may contain vulnerabilities and thus identify the vulnerable code functions or lines. Last but not least, we apply our approach to the Ethereum smart contracts written in the Solidity programming language. We choose smart contracts from distributed blockchains [209] as they become increasingly popular in various domains that involve payments and contracts. Different techniques are essential to detect their potential bugs and ensure correct executions of the payments and contracts. In short, our approach enables novel multi-level graph embeddings for fine-grained detection of smart contract vulnerabilities, and thus we name it as MANDO. MANDO is novel in its graph neural network structure that fuses topological GNN and node-level attentions with heterogeneous GNN to generate both node-level and graph-level embeddings that can capture structural information of graphs more accurately. It is also novel in enabling both node-level and graph-level classifications to detect fine-grained line-level vulnerabilities in smart contract source code in addition to coarse-grained contract-level vulnerabilities.

For our empirical evaluation, we have curated a mixed dataset containing 493 Solidity vulnerable contracts from multiple data sources from previous studies. There are seven types of vulnerabilities in the dataset; each has between 50 to 80 instances. Our evaluation results show that MANDO achieves a heightened F1-score from 81.98% to 90.51% for detecting the vulnerabilities at the fine-grained line-level, while previous deep learning and embedding-based techniques can only detect the vulnerabilities at the contract file/function level.

To summarize, our main contributions are as follows:

- We propose a new technique for representing Ethereum smart contracts written in Solidity as *heterogeneous contract graphs* that combines heterogeneous control-flow graphs (CFGs) and call graphs (CGs) of multiple levels of granularity. This new technique allows us to represent the semantic relation of node and edge types that the previous approaches could not capture with only using the homogeneous forms of these CFGs and CGs separately.

- We propose a novel architecture for the Heterogeneous Graph Neural Network using Node-Level Attention (Figure 5.2 and 5.3), which fits our customized metapaths, to build embeddings of multiple granularity levels for heterogeneous contract graphs.

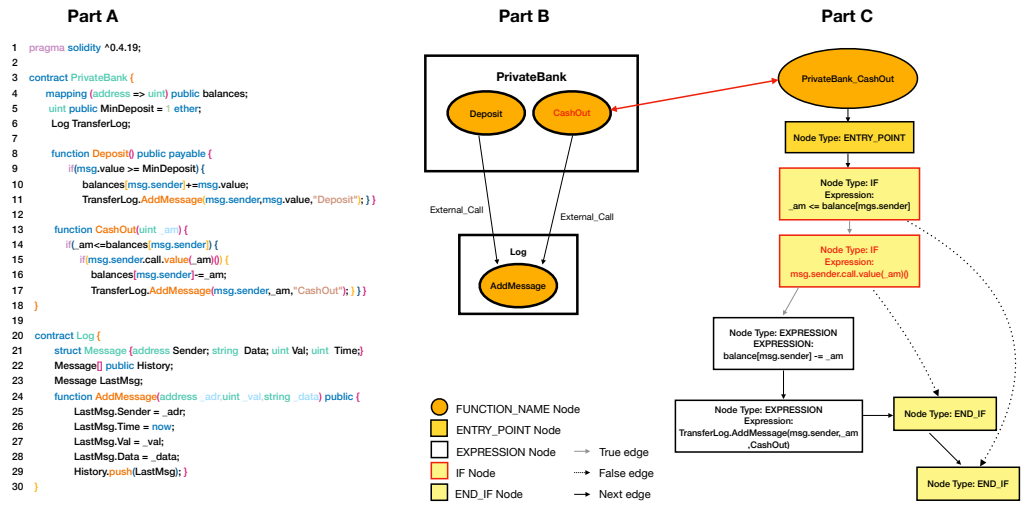


FIGURE 5.1: A sample Ethereum smart contract code snippet (Part A), its corresponding heterogeneous call graph (CG) (Part B), and a sample heterogeneous control-flow graph (CFG) for the function `CashOut` in the contract `PrivateBank` (Part C). Line 15 in Part A is the root cause of a Reentrancy bug; the nodes in CG and CFG containing the Reentrancy bug are highlighted with red text

.

- We employ the multi-level embeddings of heterogeneous graphs and labeled instances of vulnerable smart contracts to detect new vulnerabilities accurately at the line-level and contract-level, achieving better results than prior state-of-the-art bug detection techniques for smart contracts.
- We also publicize the dataset and our graph embedding models for the research community[1].

5.2 Motivation and Problem Definition

Motivating Example: Figure 5.1 (Part A) shows a sample code snippet of a smart contract written in Solidity. Part B shows the corresponding call graph (CG) of the contract. Part C shows a partial sample control-flow graph (CFG) for the `CashOut` function containing a vulnerability whose root cause is at Line 15 as `msg.sender.call` can repeatedly trigger calls to `CashOut` before `balances` is deducted at Line 16, which means `msg.sender` can receive more values than what is specified by `_am`. In order to catch this so-called *reentrance* vulnerability, the control-flow and call relations among `msg.sender`, `balances`, and `_am` should be considered. We aim to automatically capture such vulnerabilities' properties via our new graph embedding techniques.

Problem Statement: Our high-level problem is to develop more effective heterogeneous graph learning techniques, and use them to detect fine-grained

[1] https://github.com/MANDO-Project/ge-sc

line-level software vulnerabilities and their types. More specifically, our objective for smart contracts written in Solidity based on our unique graph representation and embedding techniques is to: (1) Represent it as a *heterogeneous contract graph* that combines its control-flow graph and call graph like the example in Figure 5.1; (2) Learn the embeddings of the graphs and the nodes at multiple levels of granularity to capture the syntactical and semantic information of smart contract code; (3) Accurately identify the nodes that contain certain types of vulnerabilities and locate them in the contract code.

Usage Scenarios: Such accurate vulnerability detection can be useful for smart contract quality assurance under various situations. For example:

- During the contract development in an Integrated Development Environment (IDE), it can help to identify early if the contract contains any vulnerability of known types.
- When a developer is reusing a contract from a third party, the vulnerability detection can check if it contains any known vulnerabilities and warns the developer about potential risks in reusing the contract directly.
- Whenever a new type of vulnerability is discovered, we may want to audit all existing contracts again to check if they contain the new type of vulnerability. The vulnerability detection can then be easily applied to all the contracts on a large scale for this purpose.

We believe that MANDO can be adapted to other software as long as their control-flow and call graphs can be constructed and there are vulnerability datasets available for training.

5.3 The MANDO Approach

5.3.1 Overview

This section gives an overview of our proposed approach consisting of four main components presented in the four grey boxes in Figure 5.2 and describe each component in the following subsections. The input of our approach is the source code of one or many Ethereum smart contract source files written in Solidity. The output is the bug prediction and the bug line in the source code if there is one.

First, the source code is processed by the **Heterogeneous Contract Graph Generator** component and translated into two heterogeneous graphs based on call graphs and control-flow graphs corresponding to two levels of granularity: contract level and statement (line) level, respectively. Then, the two heterogeneous graphs are fed into the second component: **Multi-Metapaths Extractor**. Based on the type of each node and the types of its associated edges, the component extracts their corresponding *metapaths*. This component is novel in the sense that it can handle dynamic numbers of node and edge types in metapaths from the automatically generated heterogeneous contract graphs. The third component, **Multi-Level Graph Neural Networks**, contains two

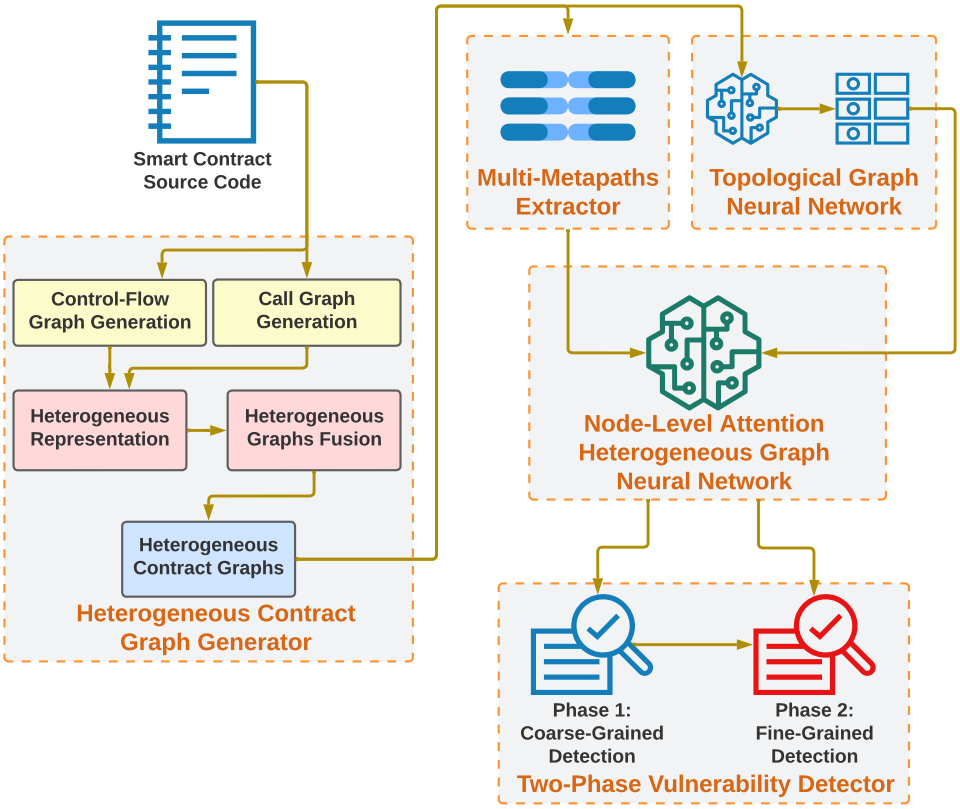


FIGURE 5.2: Overview of the MANDO framework.

steps. The first step takes metapaths or graph topology of the contract graphs from the previous component as input and generates node embeddings. Then, in the second step, the node embeddings are used as node features and fused with metapaths using heterogeneous attention mechanisms at the node level. **Two-Phase Vulnerability Detector**, the last component, uses the embeddings to train multi-layer perceptron (MLP) to perform either graph classification or node classification, depending on the kind of the input heterogeneous contract graphs. In **Coarse-Grained Detection**, the heterogeneous contract graphs embeddings are used to classify graphs if their respective contract is clean or vulnerable. In **Fine-Grained Detection**, the heterogeneous contract graphs embeddings of the vulnerable contracts, classified in the first phase, are used to classify a node of a contract graph as to whether it is clean or vulnerable. The classified nodes can then be used to find the exact locations of the vulnerabilities in specific contracts (i.e., contract-level) and specific statements or lines of code (i.e., line-level).

5.3.2 Heterogeneous Contract Graph Generator

Our approach uses Slither [123] to traverse and analyze the source code of each Ethereum smart contract for generating the basic control-flow graphs and call graphs with homogeneous structures where nodes and edges have no types or labels. Following the Definition 2.1.2.2 of a heterogeneous graph

in Section 2.1.2, we then transform these constructed graphs into heterogeneous forms to represent the semantics of graph structures and the relation of different node and edge types:

Heterogeneous Control-Flow Graphs (HCFGs). A control-flow graph of a function is an intermediate representation of all possible sequences of statements or lines of code that might be traversed when the function is executed, which is widely used in program analysis methods. Recent approaches on smart contract vulnerability detection use such graph representations of code when applying graph neural networks [76, 163], but they mostly normalize and convert those representations into homogeneous graphs before applying graph models. In particular, they only keep the major nodes and eliminate some normal nodes to normalize graphs since using nodes of diverse code semantics brings difficulties in training their graph neural networks. Thus, these approaches tend to lose valuable information regarding the source code semantics in smart contracts. In contrast, MANDO focuses on retaining most of the structure and semantics of the source code through heterogeneous representations where a variety of node types and edge types are preserved, called *heterogeneous control-flow graphs*.

The set of all node types in control-flow graphs is denoted as A_{CF}. Some typical node types include ENTRY_POINT, EXPRESSION, NEW VARIABLE, RETURN, IF, END_IF, IF_LOOP, and END_LOOP. Additionally, diverse types of connections among nodes are used to describe statements' sequential or branching structure through edge types such as NEXT, TRUE, FALSE. The set of all edge types in control-flow graphs is R_{CF}. Figure 5.1 (Part C) shows a sample heterogeneous control-flow graph generated for the *CashOut* function of contract PrivateBank. A Solidity parser (e.g., Slither) produces the complete sets of A_{CF} and R_{CF} based on the grammar of the Solidity language. $G_{CF} = \{V_{CF}, E_{CF}, \phi_{CF}, \psi_{CF}\}$ denotes an HCFG with V_{CF} and E_{CF} as its vertex and edge sets, respectively. Each node $i \in V_{CF}$ can be viewed as a tuple of (i, ϕ^i_{CF}), where i is the index of node and $\phi^i_{CF} \in A_{CF}$ is the type of node i. Similarly, each edge $(i, j) \in E_{CF}$ has an edge type $\psi^{i,j}_{CF} \in R_{CF}$. Each function in a smart contract can have an HCFG generated for it, and the HCFG has an entry node corresponding to the entry point/header of the function. A smart contract may be viewed as a set of HCFGs as it may contain more than one function.

Heterogeneous Call Graphs (HCGs). Call graphs are an intermediate representation of invocation relations among functions from the same smart contract or different smart contracts. A call graph generated via static program analysis often represents every possible call relation among functions in a program. Our study focuses on two major types of calls in smart contracts: *internal calls* for function calls inside one smart contract and *external calls* for function calls from a contract to others, represented by the two respective edge types INTERNAL_CALL and EXTERNAL_CALL. In addition, Solidity fallback functions are important in Ethereum blockchain, executed when a function identifier does not match any of the available functions in a smart contract or if no suitable data was provided for the function call. Many vulnerabilities in Ethereum smart contracts are directly or indirectly related to

such fallback functions [210]. Therefore, we represent such fallback functions with a particular node type, called FALLBACK_NODE, besides the typical function node type FUNCTION_NAME.

One HCG is generated from each smart contract. $G_C = \{V_C, E_C, \phi_C, \psi_C\}$ denotes a heterogeneous call graph with V_C and E_C as its node and edge sets, respectively. Each node i in V_C can be viewed as a tuple (i, ϕ_C^i) where i is the index of node, $\phi_C^i \in A_C$ is the type of the node i and A_C is the set of all node types in G_C. Similarly, each edge $(i, j) \in E_C$ has an associate edge type $\psi_C^{i,j} \in R_C$.

Heterogeneous Contract Graphs: Fusion of Heterogeneous Call Graphs and Heterogeneous Control-Flow Graphs. The structures of these two graphs for a smart contract can be shared or combined into a global graph to enrich information for learning. In MANDO, we design a core for HCGs and HCFGs fusion. Accordingly, the HCG edges of the smart contract act as bridges to link the discrete HCFGs of the smart contract functions into a global fused graph. Specifically, the fusion graph of the heterogeneous CG and the heterogeneous CFGs for a smart contract is denoted by $G_{Fusion} = \{V_F, E_F, \phi_F, \psi_F\}$, where $V_F = V_C \cup V_{CF}^1 \cup ... \cup V_{CF}^N$ and $E_F = E_C \cup E_{CF}^1 \cup ... \cup E_{CF}^N$, and N is number of the HCFGs for the contract. Intuitively, for each and every function node i in the call graph, the function control-flow graph G_{CF}^i is attached to the function node i at the entry node of G_{CF}^i, and thus the call graph is expanded with control-flow graphs to produce the heterogeneous contract graph. For example, in Figure 5.1, the red arrow between `CashOut` in Part B and `PrivateBank_CashOut` in Part C indicates a sample fusion between CGs and CFGs.

5.3.3 Multi-Metapaths Extractor

The number of node types in our generated graphs is dynamic and can reach sixteen, with three distinct connection types per node type, especially in the heterogeneous control-flow graphs. Pre-defining all possible metapaths with any length according to all possible node types and edge types is a challenge, as it would lead to an exponential explosion of metapaths, increased data sparsity, and reduced training accuracy. For example, in Figure 5.1, between a node of ENTRY_POINT type and a node of EXPRESSION type, several different node types can be included, such as IF and END_IF, and in other smart contracts, NEW_VARIABLE, IF_LOOP, and END_LOOP can also be included. Besides, the order of these node types can change dynamically, depending on the input contracts' structures.

In order to address the problem of exploding and changing metapaths, our method focuses on length-2 metapaths through reflective connections between adjacent nodes to extract multiple metapaths. For instance, the relation between two adjacent nodes of the types ENTRY_POINT and IF in Figure 5.1 can be described by a length-2 metapath: $ENTRY\_POINT \xrightarrow{next} IF \xrightarrow{back} ENTRY\_POINT$. HCFGs are mostly tree-like, having very few of their own back-edges induced by the LOOP-related statements in the source

code. This can lead to the lack of metapaths connecting many leaf-node types in the graphs. Adding the "back" relations helps alleviate the lack and improves the completeness of the extracted metapaths.

Previous studies [11, 211] also used length-2 in their evaluation, and a length-N metapath can be decomposed into (N - 1) length-2 metapaths. Thus, we follow those studies by using length-2 to capture the unique semantic between each node types pair and their neighbors and leave longer metapaths for future evaluations. Similar to the methods used in HAN [11], we extract the set of length-2 metapaths of each node types pair in a smart contract.

5.3.4 Multi-Level Graph Neural Networks

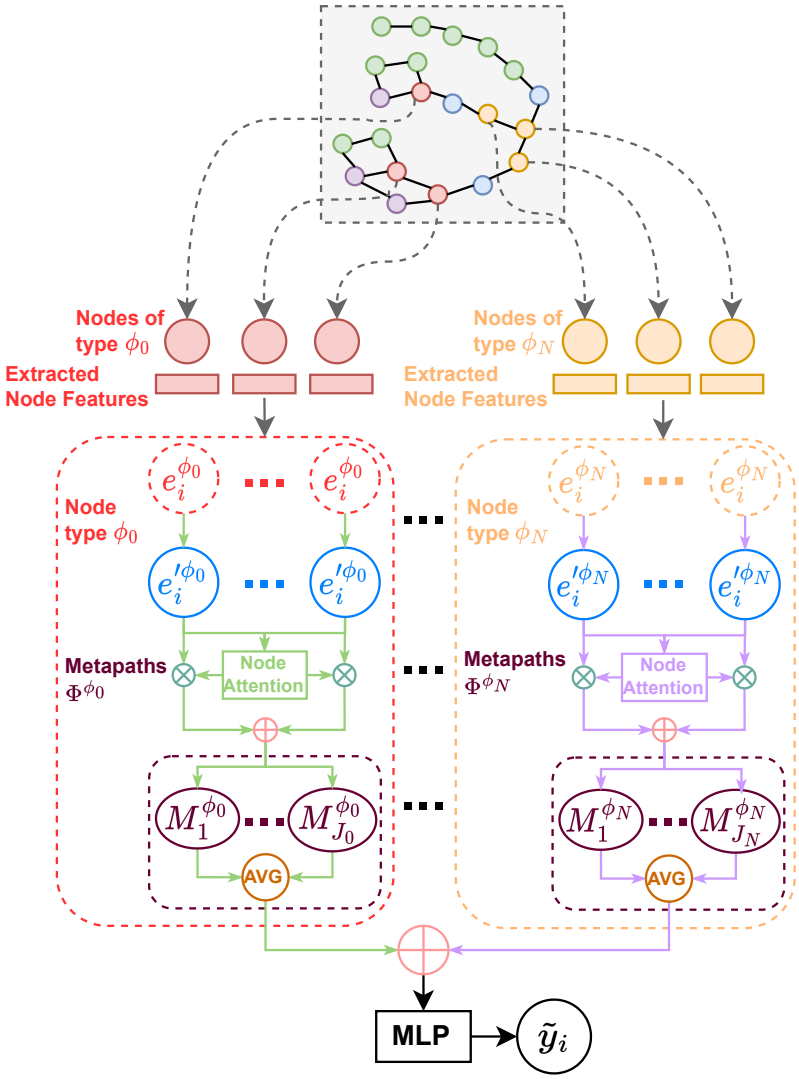


FIGURE 5.3: Our Novel Architecture for Node-Level Attention Heterogeneous Graph Neural Network in the MANDO Framework.

This component has two major building blocks: *Topological Graph Neural Network* and *Node-Level Attention Heterogeneous Graph Neural Network*. The

Notation	Explanation
i	Node i
ϕ_k	Node type k
$e_i^{\phi_k}$	Node embedding of i whose type is ϕ_k
$e_i'^{\phi_k}$	Linear transformation of $e_i^{\phi_k}$
W_{ϕ_k}	Matrix transformation for node i with type ϕ_k
$\Phi_t^{\phi_k}$	t-th metapath of node type ϕ_k
$M_{i_t}^{\phi_k}$	t-th metapath embedding of node i whose node type is ϕ_k
$M_i^{\phi_k}$	Embedding of node type ϕ_k of node i
N^{ϕ_k}	A set of metapath of node type ϕ_k
J_k	Total index of node type ϕ_k

TABLE 5.1: Table of Notation.

former learns an input graph topology, while the latter weights the importance of the metapaths in the graph.

Topological Graph Neural Network

The main goal of this building block is to capture the graph topology. Each node i has a node embedding e_i such that e_i and the embedding vector e_j of the neighboring nodes j of i are near in the embedding space. Various state-of-the-art neural network techniques can be used to generate node embeddings of graphs. For a more comprehensive comparison of their effectiveness, we employ both embedding techniques for homogeneous graphs (e.g., node2vec [6]) and embedding techniques for heterogeneous graphs (e.g., metapath2vec [41]) in our empirical evaluation (see Section 5.4).

Node-Level Attention Heterogeneous Graph Neural Network

There are two kinds of input sources for this building block: the node embeddings from the previous topological graph neural network and the metapaths from the Multi-Metapaths Extractor.

Node-Level Attention Graph Neural Network. Inspired by the node-level attention mechanism proposed by HAN [11], we also learn to weigh the importance of every metapath and node. However, unlike HAN, our novel approach can handle multiple dynamic customized metapaths without pre-defining the list of input metapaths (see Figure 5.3 and the summary of notations in Table 5.1). The previous topological graph neural network produces a node embedding $e_i^{\phi_k}$ for each node i whose type is ϕ_k; then, we construct a corresponding weighted node feature $e_i'^{\phi_k}$ by the following linear transformation:

$$e_i'^{\phi_k} = W_{\phi_k} e_i^{\phi_k}, \tag{5.1}$$

where W_{ϕ_k} is the transformation matrix associated to the type ϕ_k of node i. Each node type ϕ_k has a specific matrix W_{ϕ_k} to increase the flexibility of the transformation by projecting each type into a separated weight space.

We measure the weight of the t-th metapath $\Phi_t^{\phi_k}$ according to the node type ϕ_k of (i, j) pair by leveraging the self-attention mechanism [212] between i and j. The weight $a_{ij}^{\Phi_t^{\phi_k}}$ is defined as follows:

$$a_{ij}^{\Phi_t^{\phi_k}} = \text{softmax}_j(\text{ATT}([e_i'^{\phi_k}, e_j'^{\phi_k}]; \Phi_t^{\phi_k})), \tag{5.2}$$

where ATT is a multi-layer perceptron [213] whose values of parameters are automatically learned through back-propagation. The input of such perceptron is the concatenation of two vectors $e_i'^{\phi_k}$ and $e_j'^{\phi_k}$. We then normalize the output of ATT into the range between 0 and 1 by all neighbors of j in metapaths. The t-th metapath embedding $M_{i_t}^{\phi_k}$ of node i whose node type is ϕ_k is a weighted sum of the node features of its neighbors with corresponding weights defined in Equation (5.2). The formula is as follows:

$$M_{i_t}^{\phi_k} = \sigma\left(\sum_{j \in \mathcal{N}_i^{\Phi_k}} a_{ij}^{\Phi_t^{\phi_k}} \cdot e_j'^{\phi_k}\right), \tag{5.3}$$

where σ is the activation function, and $\mathcal{N}_i^{\Phi_k}$ denotes the neighbors of the node i according to the metapath Φ_k.

To overcome the obstacle of high variance of data in heterogeneous graphs, we propose to aggregate multi-metapath embeddings with different types of nodes. Particularly, the metapath embedding $M_{i_t}^{\phi_k}$ of each node in Equation (5.3) is calculated N times and then concatenated to create a final embedding $M'^{\phi_k}_{i_t}$ for each metapath.

After extracting the metapath embedding, we calculate the corresponding embedding of node i by averaging all metapath embedding related to i, noted AVG in Figure 5.3. Specifically, the embedding of node i with node type ϕ_k is:

$$M_i^{\phi_k} = \frac{\sum_t M_{i_t}^{\phi_k}}{|N^{\phi_k}|}, \tag{5.4}$$

where N^{ϕ_k} is a set of metapaths of the node type ϕ_k, and the total index of the node type ϕ_k is equal to the size of this set i.e., $|N^{\phi_k}|$.

For fine-grained detection, we concatenate all node embedding $M_i^{\phi_k}$ corresponding to all node type ϕ_k of all node i to generate a unified embedding vector for a node. We get the average of all node embeddings belonging to the graph for coarse-grained detection.

Optimization for Detection

We employ the multi-layer perceptron (MLP) with a softmax activation function for the graph and node classification tasks. The input of such a layer is dependent on the type of prediction tasks. The loss function for the training

process is cross-entropy, and the parameters of our model are learned through back-propagation.

5.3.5 Two-Phase Vulnerability Detector

This component has two main phases: *Coarse-Grained Detection* and *Fine-Grained Detection*. The first phase classifies clean versus vulnerable smart contracts at the coarse-grained contract level; the second phase identifies the actual locations of the vulnerabilities in the smart contract source code at the fine-grained line level. Providing line-level locations of the vulnerabilities is one of our primary contributions, while the previous graph learning-based methods [76,77] only report vulnerabilities at the contract or function level.

Phase 1: Coarse-Grained Detection

This phase classifies if a smart contract contains a vulnerability. We use the fused heterogeneous call graphs and control-flow graphs (i.e., heterogeneous contract graphs) and their embeddings to represent each input smart contract, and train the MLP (Section 5.3.4) to predict clean or vulnerable contracts. As there can be many clean smart contracts, this classification assists in reducing the search space by filtering out those clean contracts and reducing noisy data before the second phase of fine-grained vulnerability detection at the line level.

Phase 2: Fine-Grained Detection

For the vulnerable smart contracts identified in the first phase, we apply node classification on the node embeddings of their Heterogeneous Contract Graphs to identify the nodes that may contain vulnerabilities, which correspond to statements or lines of code and allow us to detect the locations of the vulnerabilities at the fine-grained line level in smart contract source code.

5.4 Empirical Evaluation

This section presents our experimental settings and results to answer these research questions: **RQ1:** The performance of our models compared to several state-of-the-art baselines on contract-level vulnerability classification, and **RQ2:** The performance of our models on line-level vulnerability detection.

5.4.1 Datasets

Our evaluation is carried out on a mixed dataset from three datasets mentioned in Section 2.2.1: **(1) Smartbugs Curated** [73,74], **(2) SolidiFI Benchmark** [75], and **(3) Clean Smart Contracts from Smartbugs Wild** [73,74]. To ensure consistency in the evaluation, we only focus on the seven types of

vulnerabilities that are joint in both datasets Smartbugs Curated and SolidiFI Benchmark, including *Access Control*, *Arithmetic*, *Denial of Service*, *Front Running*, *Reentracy*, *Time manipulation*, and *Unchecked LowLevel Calls*. Besides, based on the results of eleven integrated detection tools, the Smartbugs framework reports 2,742 contracts that do not contain any bugs, out of the 47,398 contracts in the Smartbugs Wild. Thus, we use the 2,742 contracts as the set of clean contracts.

For the coarse-grained contract-level vulnerability classification tasks, we randomly take some smart contracts from the clean set and then mix them with the Smartbugs Curated and SolidiFi Benchmark sets. We keep a ratio of 1:1 between clean and buggy contracts since this helps us create more balanced train/test sets for the tasks since there are only 44 to 95 buggy contracts labeled per each bug type (see Table 5.3). For the fine-grained line-level vulnerability detection tasks, we use the dataset containing vulnerable smart contracts only, i.e., the union of SmartBugs Curated and SolidiFI Benchmark sets. We do not use other datasets such as the ones of Zhuang *et al.* [76], Liu *et al.* [77] and eThor [78] because they do not have fine-grained line-level labels for the vulnerabilities.

Note that the Slither parser we use does not automatically generate the clean or vulnerable labels for a node. Instead, the nodes are labeled based on the lines of vulnerable code either manually by Smartbugs authors or injected by the SolidiFI tool. For example, Line 15 in Figure 5.1 contains a Reentrancy bug labeled by Smartbugs; then, the nodes with red text in the Heterogeneous CFG and CG are labeled vulnerable.

5.4.2 Comparison Methods

Comparison to Graph-based neural network Methods

We use the four state-of-the-art methods presented in Section 2.1.2, including*node2vec* [6], *LINE* [7], *GCN* [8], and *metapath2vec* [41]. Note that the original architectures of node2vec, LINE, GCN, and metapath2vec only focus on graph topology and do not have any components to handle node features.

Although HAN [11] inspired some idea for our Node-Level Attention Heterogeneous Graph Neural Network, our approach has novelty in resolving the challenges of fitting with the customized metapaths that the original HAN model could not handle effectively. In particular, HAN requires a predefined list of metapaths and each HAN model only serves one or some predefined node types. However, the MANDO's Heterogeneous CFGs and CGs have dynamic types of nodes and edges, leading to difficulties in predefining metapaths like the original HAN model, and thus we did not use HAN as a baseline in our evaluation.

The output embeddings of the homogeneous and heterogeneous graph neural networks are used in two ways in our evaluation. First, we use them directly as the baselines for the coarse-grained graph classification tasks and fine-grained node classification tasks. Second, each of the graph neural networks is plugged into MANDO as the topological graph neural network.

The generated embeddings are then considered as the node features fed to MANDO's Node-Level Attention Heterogeneous Graph Neural Network. Besides, we use fully-connected layers as the multi-layer perceptron in node and graph classification tasks. In addition, the one-hot vectors based on the Node-Type is also used as the node features, which allows MANDO to perform independently without relying on any added-in topological graph neural network.

Parameter Settings: The node embedding size is set to 128 for all models. We use an adaptive learning rate from 0.0005 to 0.01 in coarse-grained tasks and from 0.0002 to 0.005 in fine-grained tasks when training. For each GAT layer [9] of each metapath that feeds to the MANDO's Self-Attention Layer per Node Type, we set 8 multi-heads whose hidden size is 32. The numbers of learning epochs of coarse-grained and fine-grained tasks are 50 and 100, respectively, to reach converging. For node2vec, LINE, GCN, and metapath2vec, we use the authors' recommended settings to ensure the highest performance.

Comparison with Conventional Detection Tools

We also compare our method to six common smart contract vulnerability detection tools based on traditional software engineering approaches: *Manticore* [117] analyzes the symbolic execution of smart contracts and binaries; *Mythril* [214] uses symbolic execution, SMT solving, and taint analysis to find out the security vulnerabilities of smart contracts; *Oyente* [127] analyzes symbolic execution to detect bugs in the Ethereum blockchain; *Securify* [126] can prove if the behavior of a smart contract is safe or not according to given predicates and by checking its graph dependencies; *Slither* [123] reduces the complexity of instruction sets with the intermediate representation of Ethereum smart contract called SlithIR, while retaining much of the semantic to increase the accuracy of bug detection; *Smartcheck* [128] converts smart contracts into XML-based representation and finds possible bugs along executive paths.

5.4.3 Evaluation Metrics

Since our prediction results are based on the binary classification of a node or a graph, we use F1-score and Macro-F1 scores to measure the prediction performance. The former is a measure of a model's performance by balancing between precision and recall, while the latter is used to assess the quality of problems with multiple binary labels or multiple classes. In our evaluation, the F1-score metric is used to evaluate the models' performance when finding vulnerabilities in the graphs, and we also call it *Buggy-F1*. Macro-F1 is considered to avoid biases in the clean and vulnerability labels.

Methods		Metrics	Access Control	Arithmetic	Denial of Service	Front Running	Reentrancy	Time Manipulation	Unchecked Low Level Calls
Heterogeneous GNN	metapath2vec	Buggy F1	62.90%	56.46%	55.17%	63.40%	61.79%	66.29%	55.22%
		Macro-F1	42.55%	46.32%	44.49%	43.03%	47.26%	45.94%	49.05%
Homogeneous GNNs	GCN	Buggy F1	60.63%	-	60.12%	-	-	59.60%	-
		Macro-F1	48.45%	-	45.65%	-	-	46.60%	-
	LINE	Buggy F1	61.45%	33.41%	59.61%	62.61%	66.23%	66.65%	60.51%
		Macro-F1	40.88%	33.47%	35.77%	34.29%	37.91%	40.84%	40.08%
	node2vec	Buggy F1	62.63%	58.59%	56.41%	64.77%	58.29%	63.03%	61.69%
		Macro-F1	48.83%	50.80%	40.63%	46.08%	45.80%	46.78%	49.91%
MANDO with Node Features Generated by	**NodeType One Hot Vectors**	Buggy F1	**71.19%**	**66.85%**	87.37%	87.31%	**76.09%**	85.03%	**72.08%**
		Macro-F1	**74.57%**	**71.04%**	86.68%	85.65%	**75.80%**	83.35%	**74.52%**
	metapath2vec	Buggy F1	57.70%	52.84%	60.16%	62.19%	55.06%	59.47%	51.37%
		Macro-F1	55.60%	55.06%	64.12%	64.80%	60.96%	57.74%	55.58%
	GCN	Buggy F1	49.26%	-	53.19%	-	-	49.50%	-
		Macro-F1	52.75%	-	60.26%	-	-	57.31%	-
	LINE	Buggy F1	65.12%	54.91%	**89.15%**	**89.86%**	71.04%	**87.71%**	59.44%
		Macro-F1	70.15%	65.36%	**89.46%**	**88.66%**	74.97%	**86.41%**	66.16%
	node2vec	Buggy F1	55.71%	64.11%	83.86%	86.05%	71.39%	73.38%	66.10%
		Macro-F1	64.70%	70.23%	83.40%	84.95%	72.31%	74.36%	71.02%

TABLE 5.2: Average Performance Comparison of the Coarse-Grained Contract-Level Detection over 20 Runs. We use the *Heterogeneous Contract Graphs* of both Clean and Buggy Smart Contracts as the MANDO framework inputs. *Buggy- F1* means the F1-score of the buggy graph label. '–' denotes not applicable due to the insufficiency of GPU memory to handle the input graphs for the GCN model.

Bug Types	# Total / Buggy Contracts	# Total Nodes	# Total Edges	# Buggy Nodes
Access Control	114 / 57	13014	10721	7500
Arithmetic	120 / 60	17372	14271	10110
Denial of Service	92 / 46	13968	11997	8280
Front Running	88 / 44	22824	19761	10008
Reentrancy	142 / 71	18898	17614	11238
Time Manipulation	100 / 50	16765	15550	10051
Unchecked Low Level Calls	190 / 95	17756	14858	7583

TABLE 5.3: Statistics of the Mixed Dataset.

5.4.4 Empirical Results

Table 5.3 shows the statistics of the mixed dataset. In the initial experiments, we split the dataset into 60% / 20% / 20% for the corresponding train / validation / test sets. However, some bug types in our mixed dataset have less than 100 contracts, which leads to a lack of enough samples for training. Besides, we realized that the loss value remains stable after a fixed number of epochs (100 and 50 epochs for Fine-Grained for Coarse-Grained tasks, respectively). Hence, we decided to split the dataset to 70%/30% to increase the train/test set sizes and maintain the vulnerable nodes' ratio in each set corresponding to the whole dataset. To get robust results for each dataset, each embedding method, and each vulnerability type, we run the experiment twenty times independently, each time with a different random seed, and report the average results. Besides, our approach shows impressive capabilities in training and inference time. It takes around 30 seconds for over ten thousand nodes and edges in the node classification task and under 10 seconds for about 100–200 contracts in the graph classification task. Also, it requires under 1 second for

all inferences.

Coarse-Grained Contract-Level Vulnerability Detection (RQ1)

In this experiment, we want to measure MANDO's performance with various node feature generator components in detecting vulnerable smart contracts (see Section 5.3.5). It illustrates the flexibility of our method working with different graph neural networks. Table 5.2 presents MANDO's performance via several different graph neural methods on various vulnerability types. Accordingly, we have some observations:

- MANDO generally outperforms baseline GNNs in contract-level detection. For instance, the Buggy-F1 and Macro-F1 of MANDO are over 88.66%, while the maximum performance of the baselines is 64.77% in detecting the Front-Running vulnerability type.
- It is unclear which node feature generation method is the best among the heterogeneous and homogeneous GNNs and the node-type one-hot vectors. However, integrating these types of GNNs inside MANDO outperforms all the baselines. Hence, we believe that the architecture of MANDO for combining different GNNs is suitable for classifying vulnerable smart contracts.
- MANDO is reliable in determining whether an unknown smart contract contains vulnerabilities, especially for the vulnerability types of Denial of Service, Front Running, and Time Manipulation with Buggy-F1 over 87.7%. MANDO is highly compatible with different solidity versions based on the Slither tool [123], and its trained models can be applied in practice to audit newly-appeared smart contracts that previous studies using graph learning [76, 163] have not been able to do effectively (see Section 2.2.3).

Fine-Grained Line-Level Vulnerability Detection (RQ2)

To help smart contract developers to locate vulnerabilities more easily, vulnerability detectors should be able to identify the vulnerabilities at the more fine-grained line level (see Section 5.3.5). In this experiment, we examine the performance of our method with respect to various state-of-the-art methods for line-level detection.

Table 5.4 shows the performance of our method trained with different models for Topological Graph Neural Network and the baselines methods, including graph-based neural networks and the conventional detection tools based on various software engineering techniques. From the table, we observe:

- Generally, MANDO outperforms conventional detection tools significantly. Remarkably, an improvement is up to 63.4% of MANDO compared to the best performance of the tools in detecting Reentrancy bugs. We argue the significant improvement is from two sources: First, our constructed heterogeneous graphs retain more CFGs' aspects than other

Methods		Metrics	Access Control	Arithmetic	Denial of Service	Front Running	Reentrancy	Time Manipulation	Unchecked Low Level Calls
Conventional Detection Tools	securify	Buggy F1	13.0%	0.0%	18.0%	53.0%	23.0%	24.0%	11.0%
		Macro-F1	52.3%	45.2%	52.0%	72.2%	58.4%	52.4%	54.1%
	mythril	Buggy F1	34.0%	73.0%	41.0%	63.0%	19.0%	23.0%	14.0%
		Macro-F1	61.1%	**84.1%**	60.1%	77.8%	55.3%	50.8%	55.7%
	slither	Buggy F1	32.0%	0.0%	13.0%	26.0%	15.0%	44.0%	10.0%
		Macro-F1	61.5%	45.2%	42.7%	56.9%	49.4%	57.3%	53.3%
	manticore	Buggy F1	30.0%	30.0%	12.0%	7.0%	9.0%	24.0%	4.0%
		Macro-F1	61.1%	61.0%	48.0%	46.9%	51.2%	55.1%	50.6%
	smartcheck	Buggy F1	20.0%	22.0%	52.0%	0.0%	22.0%	44.0%	11.0%
		Macro-F1	56.0%	56.1%	69.9%	46.2%	57.8%	64.2%	54.1%
	oyente	Buggy F1	21.0%	71.0%	48.0%	0.0%	20.0%	24.0%	8.0%
		Macro-F1	57.3%	82.8%	67.2%	44.8%	56.1%	52.4%	52.6%
Heterogeneous GNN	metapath2vec	Buggy F1	35.46%	68.70%	60.64%	80.65%	71.66%	67.51%	26.06%
		Macro-F1	48.52%	47.08%	48.67%	49.88%	49.15%	49.00%	49.91%
Homogeneous GNNs	GCN	Buggy F1	43.92%	65.69%	64.06%	81.09%	71.76%	68.70%	38.13%
		Macro-F1	54.20%	53.42%	54.81%	56.21%	53.00%	52.74%	53.57%
	LINE	Buggy F1	53.59%	68.61%	62.28%	83.06%	74.78%	70.76%	7.10%
		Macro-F1	57.75%	48.53%	51.63%	42.27%	38.26%	42.40%	44.31%
	node2vec	Buggy F1	44.94%	67.84%	63.92%	81.84%	71.52%	67.81%	34.26%
		Macro-F1	54.73%	52.92%	54.83%	56.17%	53.45%	53.19%	53.09%
MANDO with Node Features Generated by	**NodeType One Hot Vectors**	Buggy F1	77.21%	81.62%	79.83%	88.19%	84.24%	86.64%	65.95%
		Macro-F1	74.89%	76.01%	76.22%	68.70%	75.89%	82.72%	75.01%
	metapath2vec	Buggy F1	67.97%	74.84%	67.22%	86.08%	76.03%	73.81%	50.71%
		Macro-F1	67.87%	65.92%	62.90%	65.22%	66.04%	71.04%	64.73%
	GCN	Buggy F1	69.00%	76.47%	70.88%	87.15%	77.57%	77.73%	52.95%
		Macro-F1	66.77%	66.75%	64.26%	65.71%	65.85%	73.94%	65.75%
	LINE	Buggy F1	81.19%	81.58%	**82.12%**	90.47%	86.27%	89.21%	83.37%
		Macro-F1	**80.93%**	77.80%	**79.00%**	78.43%	80.43%	86.17%	85.40%
	node2vec	Buggy F1	**81.98%**	**84.35%**	82.09%	**90.51%**	**86.40%**	**90.29%**	**84.81%**
		Macro-F1	79.23%	79.10%	77.84%	**78.60%**	**80.78%**	**86.76%**	**86.74%**

TABLE 5.4: Average Performance Comparison of the Fine-Grained Line-Level Detection over 20 Runs. We use the *Heterogeneous Contract Graphs* of the Buggy Smart Contracts as the inputs for MANDO framework. *Buggy- F1* means the F1-score of the buggy node label. A total of fifteen methods are examined in the comparisons. The best performance in each vulnerability category is highlighted.

analysis tools. Secondly, our node-level attention module is flexible enough for GNNs to learn the exact locations of vulnerabilities within contracts.

- Our method beats the results of the baseline GNNs. Remarkably, the macro-F1 scores of the baseline GNNs are up to 60.5%, while our models can reach up to 80.78%. Hence, it is evident modeling the smart contracts as Heterogeneous Contract Graphs can benefit vulnerability prediction.

- Conventional detection tools perform well in detecting arithmetic bugs. The phenomenon is reasonable since these tools mostly use symbolic execution and such technique is suitable for detecting arithmetic bugs [215]. However, MANDO performance is still on par with the tools and our future work will improve the graph models to learn arithmetic operations better. Besides, some conventional detection tools in Table 5.4 barely work (with Buggy-F1=0%) for some vulnerability types due to their intrinsic limits in relying on predefined expert patterns that could not capture these vulnerabilities.

Expanded Experiments. We also ran the experiments in Tables 5.2 and 5.4 with only Heterogeneous CFGs and CGs separately. Overall, these results are worse than the fusion form in the heterogeneous contract graphs reported in the chapter. The expanded experiments can be found in our Git repository link.

Chapter 6

Heterogeneous Graph Transformers for Smart Contract Vulnerability Detection on Source Code and Bytecode

Smart contracts in blockchains have been increasingly used for high-value business applications. It is essential to check smart contracts' reliability before and after deployment. Although various program analysis and deep learning techniques have been proposed to detect vulnerabilities in either Ethereum smart contract source code or bytecode, their detection accuracy and scalability are still limited. As a successor of MANDO presented in Chapter 5, we propose a novel framework named MANDO-HGT for detecting smart contract vulnerabilities in this chapter. Given Ethereum smart contracts, either in source code or bytecode form, and vulnerable or clean, MANDO-HGT custom-builds *heterogeneous contract graphs* to represent control-flow and/or function-call information of the code. It then adapts heterogeneous graph transformers (HGTs) with customized meta relations for graph nodes and edges to learn their embeddings and train classifiers for detecting various vulnerability types in the nodes and graphs of the contracts more accurately. Our empirical results show that MANDO-HGT can significantly improve the detection accuracy of other state-of-the-art vulnerability detection techniques that are based on either machine learning or conventional analysis techniques. The accuracy improvements in terms of F1-score range from 0.7% to more than 76% at either the coarse-grained contract level or the fine-grained line level for various vulnerability types in either source code or bytecode. Our method is general and can be retrained easily for different vulnerability types without the need for manually defined vulnerability patterns.

6.1 Introduction

Smart contracts on blockchain systems have been used for many application domains [216], such as finance, e-commerce, healthcare, logistics, and law. Any bug or security vulnerability in a smart contract deployed in a blockchain can have devastating consequences for both the developers and the users of the smart contracts [217, 218]. Therefore, there is a high demand for various kinds of security assurance techniques for smart contracts, especially for vulnerability detection.

Many studies have been carried out on vulnerability detection in smart contracts based on conventional software testing, analysis, verification techniques [112, 117, 123, 126–128, 132, 133]. Such techniques often require certain types of oracles or specifications of the (un)expected patterns or semantics of smart contract code for analysis. Unfortunately, specifying the patterns can take much manual effort and make it troublesome to adapt the tools to the evolving contract languages and types of vulnerabilities. Also, computational complexity makes it very expensive to repeatedly run the techniques on a large set of smart contracts to search for new types of vulnerabilities. Hence, a new class of vulnerability detection techniques has been proposed based on

machine learning and deep learning techniques [76,77,149,151,165,167]. Such techniques aim to encode various syntactic and semantic code information via syntax trees, control-flow graphs, or program dependency graphs, among others, and to train automated classifiers to distinguish vulnerable code from normal ones. The learning-based techniques reduce the need for manually specified patterns or specifications, easier to be adapted to new types of code and vulnerabilities as long as some training data is provided. However, existing code learning techniques often treat most nodes and edges *homogeneously*, ignoring fine-grained differences in the nodes and edges types and their exact locations in the code's trees and graphs. This leads to insufficient learning accuracy, and this limitation becomes more pronounced for smart contracts bytecode without source code. Since the structures representing different bytecode become more similar to each other if the types of the specific bytecode instructions are ignored, making it harder to identify vulnerable/bug patterns.

Combining the advantages of previous techniques and progresses in heterogeneous graph learning [11, 12, 41, 219], this chapter aims to develop a new framework for smart contract vulnerability detection, applicable to both source code and bytecode. The main idea of our framework is two-folded:

- First, we represent the contract code, either source code or bytecode, as customized heterogeneous contract graphs that represent control flows and call relations of the code. With the combined representations, we aim to capture the code's syntactic and semantic information more comprehensively to facilitate learning of code patterns and distinguishing vulnerable code from clean ones.

- Second, we extend the heterogeneous graph transformer (HGT) [12] techniques to learn different types of nodes and edges of the contract graphs and encode the semantics of the code more accurately. The encodings can then be used to train classifiers to recognize vulnerable code.

We name our framework MANDO-HGT, following our previous work MANDO in Chapter 5 that only works for source code and uses a different graph learning technique. Our framework aims to be more general than previous techniques, applicable for either source code or bytecode, can be instantiated with various graph learning techniques, and can be re-trained for new types of vulnerabilities and detect vulnerabilities in large sets of contracts efficiently and accurately. The general approach is useful since it is not uncommon for smart contracts to be deployed *without* their source code or for the source code to be lost or deleted over time and the approach should be able to handle variation in generated bytecode to some extent when retrained with bytecode variants. Furthermore, MANDO-HGT can be re-trained to detect different types of vulnerabilities without manually defining bug patterns needed by analysis-based techniques. With its flexible design, we believe that MANDO-HGT can be integrated with other code

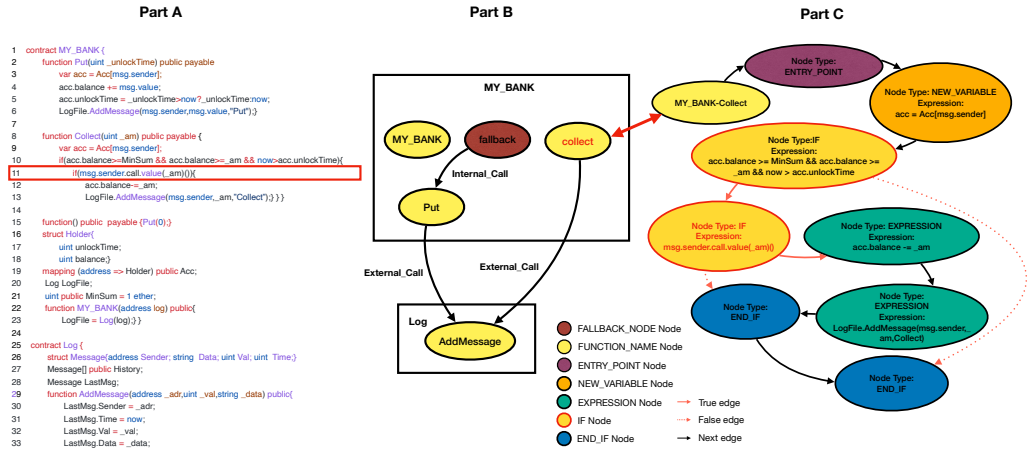


FIGURE 6.1: A sample Ethereum smart contract `MY_BANK` (Part A), its call graph (CG) (Part B), and a control-flow graph (CFG) (Part C) for the function `Collect`. Line 11 in Part A is the root cause of a *reentrancy* bug; the nodes in CG and CFG containing the *reentrancy* bug are highlighted with red text.

representation and learning techniques to improve its vulnerability detection accuracy further.

Similar to Chapter 5, we have curated 55k Ethereum smart contracts from various data sources, including SmartBugs [73,74] and SolidiFI Benchmark [75], then verified the labels for 423 buggy and 2,742 clean contracts and evaluated MANDO-HGT on the mixed dataset. Our evaluation shows that MANDO-HGT significantly improves F1-score over other vulnerability detection techniques: (1) Compared to other best-performing learning-based techniques, it improves their F1-score by 0.74% to 22.56% at the contract level and 3.51% to 7.48% at a more fine-grained line level for various vulnerability types in either source code or bytecode; (2) Compared to best-performing conventional analysis-based techniques that detect vulnerabilities at the fine-grained line level, it improves their F1-score by 18.18% to 76.89%;

We also show that, through a few case studies assisted by recent neural network interpretation techniques [220,221], the detection results of MANDO-HGT are often meaningful, reflecting our understanding of the bug patterns. We also provide possible explanations for the failures in a few cases where the detection results are wrong, which may guide future improvements to learning-based techniques.

6.2 Motivation

Motivating Sample Source Code. Similar to Figure 5.1, Figure 6.1 (Part A) presents a snippet of a smart contract written in Solidity with a *reentrance* vulnerability. Part B presents the call graph (CG) of the contract, and part C presents a partial sample control-flow graph (CFG) for the `collect` function of the sample contract. Line 11. `msg.sender.call`, is the root cause of the vulnerability of this sample code. `collect` can be repeatedly called before `balances` is deducted at Line 12, allowing `msg.sender` to receive more values

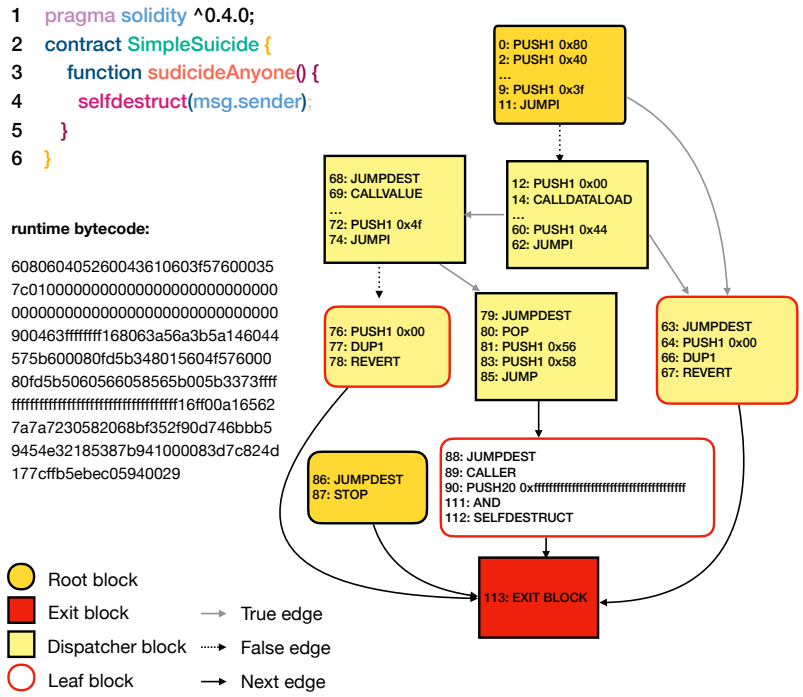


FIGURE 6.2: Code snippet, runtime bytecode, and control-flow graph of the runtime bytecode of a contract containing an *access control* bug.

than what is specified by `_am`. In order to catch this so-called *reentrance* vulnerability, the control-flow and call relations among `msg.sender`, `balances`, and `_am` should be considered.

Motivating Sample Bytecode. Figure 6.2 shows a snippet of a smart contract written in Solidity, together with its runtime bytecode [1] and the control-flow graph of the bytecode. It contains an *access control* vulnerability on lines 3–4 as `selfdestruct` is a critical function in Ethereum, leading to the self-destruction of the smart contract but inadequately protected. Thus, malicious parties can destruct the contract due to missing access controls. This vulnerability would be represented by two node-edge-type relations in our heterogeneous control-flow graph as $DISPATCHER \xrightarrow{true} DISPATCHER$ and $DISPATCHER \xrightarrow{next} LEAF$.

Objectives. Our primary objective is to automatically capture vulnerabilities in either contract source code like Figure 6.1 and contract bytecode like Figure 6.2 via our graph embedding techniques. More specifically, our objective is to: (1) Represent the Solidity source code or EVM bytecode as heterogeneous call and control-flow graphs like the example flow charts; (2) Learn the embeddings of the nodes and graphs; and (3) Efficiently and accurately identify if a contract contains vulnerabilities and locate them if its source code is available.

[1] When a contract in Solidity source code is compiled, the produced bytecode has two types: *creation* bytecode is the constructor code of the contract that performs initializations and deploys the *runtime* bytecode to the blockchain; the constructor code is then discarded, not stored in blockchain.

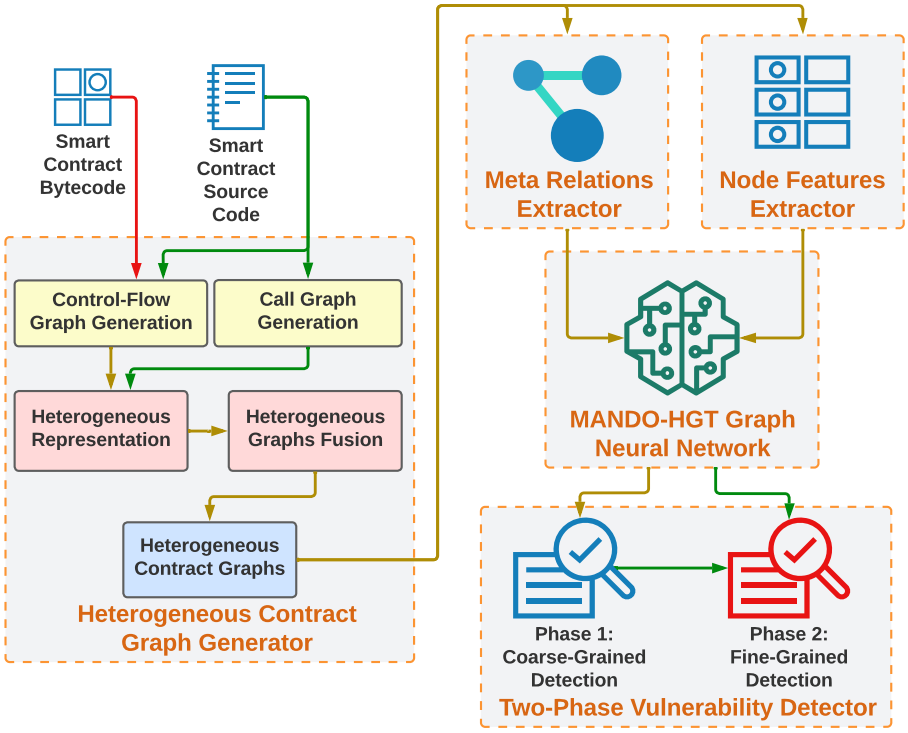


FIGURE 6.3: MANDO-HGT Overview. The process flows indicated by yellow arrows are for both the bytecode and source code of the input contracts, while green arrows are for source code only.

6.3 Approach

Our proposed framework, MANDO-HGT, consists of five main components in the grey boxes in Figure 6.3: *Heterogeneous Contract Graph Generator*, *Meta Relations Extractor*, *Node Features Extractor*, *MANDO-HGT Graph Neural Network*, and *Two-Phase Vulnerability Detector*. The five components are explained in detail below. The input of MANDO-HGT is either the source code or bytecode of one or more Ethereum smart contracts, and the output is the bug predictions for the input contracts at the contract level (for both source code and bytecode) and the line-level (for source code only).

6.3.1 Heterogeneous Contract Graph Generator

Smart contract code, either source code or bytecode, is processed by the first component of Figure 6.3, **Heterogeneous Contract Graph Generator**, and translated into heterogeneous graphs based on control-flow graphs (CFGs) and/or call graphs (CGs). In MANDO-HGT, we use Slither [123] to analyze source code and EtherSolve [222] to analyze bytecode, respectively, to construct basic CFGs and CGs. In contrast to previous studies [76, 163] that only consider *homogeneous* forms of control-flow graphs where types of nodes and edges are not utilized, we retain most of the structure and semantics of smart contract code through *heterogeneous* graphs that preserve various node and edge types. In particular, we convert basic CFGs and CGs into heterogeneous

forms, called *heterogeneous control-flow graphs* and *heterogeneous call graphs*, and fuse them into *heterogeneous contract graphs*.

Heterogeneous Control-Flow Graphs (HCFGs). For input bytecode, a CFG may involve all possible opcodes defined in the Ethereum yellow paper [209], but using all the opcodes as node types can induce much learning overhead. Based on EtherSolve [222], we define six primary types of nodes representing important opcode blocks, including *ROOT*, *BASIC*, *DISPATCHER*, *FALLBACK*, *LEAF*, and *EXIT* in MANDO-HGT: *ROOT* and *EXIT* represent entry and end blocks, respectively; *DISPATCHER*, *FALLBACK*, and *LEAF* are *BASIC* blocks with some unique characteristics. Specifically, a *BASIC* block is a sequence of opcodes executed sequentially between a jump destination (`JUMPDEST` opcode) and a jump instruction (`JUMP` or `JUMPI` opcode). A *DISPATCHER* block is a *BASIC* block with the last opcode being the return or stop opcode. A *FALLBACK* block is a *DISPATCHER* block that has no call data, and none of the hashes matches when executed (`REVERT` opcode). A *LEAF* block has the last opcodes being `REVERT`, `SELFDESTRUCT`, `RETURN`, `INVALID`, and `STOP` and has no successors, which means jumping to the *END* block in the CFG. In addition, three edge types are used to describe sequential (*NEXT*) or branching (*TRUE* and *FALSE*) connections between nodes. Notably, the `JUMPI` opcode plays a vital role in a conditional branching structure; we add the edge type *FALSE* for a branch that leads to the following opcode block when the branch condition is false, and the *TRUE* edge type is for the true branch, which is the argument of the `PUSH` opcode interpreted as the destination offset for the `JUMPI`. Thus, a smart contract can be converted to a heterogeneous control-flow graph (HCFG). Figure 6.2 shows the generated HCFG for the runtime bytecode of a buggy contract.

For input source code, a CFG may involve many types of statements or lines of code. We use typical statement types as the node types for source code's HCFGs, such as *ENTRY_POINT*, *EXPRESSION*, *NEW_VARIABLE*, *RETURN*, *IF*, *END_IF*, *IF_LOOP*, and *END_LOOP*. Similar to bytecode, three edge types are used to indicate statements' sequential or branching nature, such as *NEXT*, *TRUE*, and *FALSE*. Figure 6.1 (Part C) shows a sample HCFG generated for the `Collect` function in the `MY_BANK` contract.

Due to the capabilities and limitations of the tools for generating CFGs (Slither [123] for Solidity source code and EtherSolve [222] for EVM bytecode), a bytecode's HCFG represents the control flows throughout an entire smart contract, while a source code's HCFG is only for one function of a contract. To integrate the HCFGs for all functions of a contract, we also utilize call graphs for source code as explained below.

Heterogeneous Call Graphs. A call graph (CG) represents the invocation relations among functions in one or multiple smart contracts. There are two basic forms of calls in smart contracts that the MANDO-HGT framework considers: *internal calls* for function calls within the same contract and *external calls* for function calls across contracts, represented by two edge types *INTERNAL_CALL* and *EXTERNAL_CALL* respectively. In addition to the typical function node type *FUNCTION_NAME*, we also employ the

FALLBACK_NODE node type to represent fallback functions that are executed if a function identifier to be called does not match any accessible function in a smart contract or if insufficient data was provided for the function call. Such fallback functions are directly or indirectly related to numerous Ethereum smart contract vulnerabilities [210]. Figure 6.1 (Part B) shows such a heterogeneous call graph.

We also use Slither to process each smart contract source code to produce its heterogeneous call graph and add the explicit types to the nodes and edges.

Heterogeneous Contract Graphs: Fusion of Heterogeneous Call Graphs and Heterogeneous Control-Flow Graphs. The topologies of these two kinds of graphs for a smart contract source code can be merged into a global graph, to facilitate the graph learning process later. In MANDO-HGT, the sub-component **Heterogeneous Graphs Fusion** is for this purpose: for each node in the heterogeneous call graph that represents a function, a bridging edge is added to link this node to the entry node of the heterogeneous control-flow graph for the function (as illustrated by the double-arrow edge between Figure 6.1 Part B and Part C). We call such a fused graph a *heterogeneous contract graph*. For bytecode, since the heterogeneous control-flow graph generated by EtherSolve has represented the entire smart contract, our **Heterogeneous Graphs Fusion** sub-component directly utilizes the bytecode's HCFG as the fused heterogeneous contract graph.

6.3.2 Meta Relations Extractor

The component **Meta Relations Extractor** of MANDO-HGT extracts customized meta relations from the generated *heterogeneous contract graphs*. The main advantage of extracting meta relations is avoiding the explosion of all possible node and edge types combinations in the traditional approaches that use metapath [11, 41] as the number of node types and edge types in the graphs is dynamic and can be up to eighteen node types and five edge types.

We also add meta relations through reflective connections between adjacent nodes, e.g., the relation between two adjacent nodes of the types EXPRESSION and END_IF in Figure 6.1 can be described by both $\langle EXPRESSION, next, END\_IF \rangle$ and $\langle EXPRESSION, back, END\_IF \rangle$. Heterogeneous contract graphs are predominantly tree-like, with only a few back-edges created by LOOP-related statements. Adding the reflective relations increases the comprehensiveness of the extracted meta-relations, and improves the stable operability of the heterogeneous graph transformer (HGT) used in MANDO-HGT because the original architecture of each HGT layer requires at least two source nodes for one target node [12] and, without reflective relations, many nodes having only one source node (e.g., the two EXPRESSION nodes in Figure 6.1) would be ignored during training.

6.3.3 Node Features Extractor

The main goal of this extractor is to generate basic node features via one of the two following ways. (1) Generate node embeddings via some basic graph

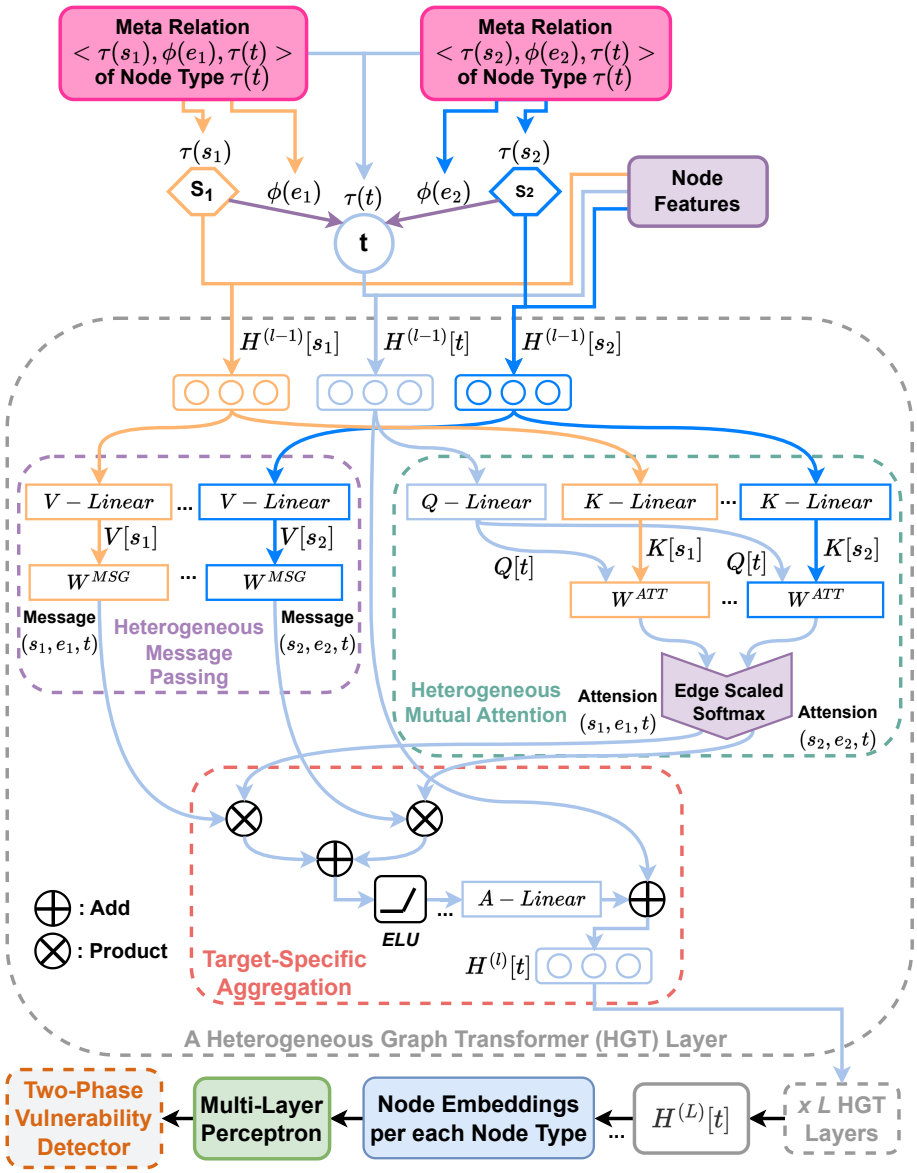


FIGURE 6.4: The architecture of the MANDO-HGT Graph Neural Network.

neural network without considering node and edge types. Omitting node and edge types is often reasonable for graph classification, as the connectivity topology among nodes is often sufficient to differentiate the graphs. We employ either homogeneous (e.g., node2vec [6]) or heterogeneous (e.g., metapath2vec [41]) graph neural networks in our evaluation (Section 6.4). (2) Generate one-hot vectors based on the node types as the node features. These node features were used as initial embeddings for the MANDO-HGT's first layer to leverage the rich information independently *without* relying on any other neural network.

6.3.4 MANDO-HGT Graph Neural Network

Figure 6.4 illustrates the architecture of the MANDO-HGT Graph Neural Network, based on Heterogeneous Graph Transformer (HGT) [12]. In MANDO-HGT GNN, we feed all pairs of meta relations of every target node, including their node types and node features, as inputs to one HGT layer. This is the major difference in our framework from the original HGT GNN. Such a mechanism allows MANDO-HGT to learn the inter-relation among our customized meta relations in heterogeneous contract graphs. It is important since it disentangles complex node/edge relations for learning. The outputs of **MANDO-HGT GNN** are fed into the final components **Two-Phase Vulnerability Detector** to identify whether the smart contracts contain bugs and to find the bug locations in the contract source code.

Heterogeneous Graph Transformer (HGT) layer. The goal of this layer is to learn the *attention* of every pair of meta relations between a target node t and its neighbor source nodes s_1 and s_2 [12]. To achieve the goal, the architecture of Transformer [212] is employed with the target node t as the "Query" vector and its neighbors s_1 and s_2 as "Key" vectors. The attention is the output of the softmax layer applied to the concatenation of the output of h attention head [9]. Each attention head explores a different relation aspect of the two pairs of t with s_1 and t with s_2 by letting the embedding vectors of t with s_1 and t with s_2 go through the l-th GNN layer denoted by $H^{(l-1)}[t]$, $H^{(l-1)}[s_1]$ and $H^{(l-1)}[s_2]$ in Figure 6.4. Specifically, there are three sub-components inside the HGT layer: *(1) Heterogeneous Mutual Attention:* The sub-component uses the Q and K linear transformations of target node t, and the two source neighbors s_1 and s_2 of t as the inputs, and the output is the attention or correlation probability of the two node pairs, i.e., s_1 or s_2 with t as well as the edge types $\phi(e_1)$ and $\phi(e_2)$ associated with the two given source nodes. The matrix W^{ATT} encodes multiple semantic relations of the pairs with the same node type. *(2) Heterogeneous Message Passing:* The input in this sub-component is the V linear transformations of the pair of source nodes s_1 and s_2, and the output is the multi-head message containing the distribution differences of nodes and edges with different types. We use matrix W^{MSG} to capture the edge dependency of each head. Note that the sub-component does not depend on the one above, so it can be processed simultaneously as the previous one. *(3) Target-Specific Heterogeneous Message Aggregation:* The sub-component is a multi-layer perception whose input is the aggregation of the outputs of the two components above, and the output is the contextualized representative vector $H^{(l)}$ of node t. Also, this sub-component uses an Exponential Linear Unit (ELU) as an activation function.

The target node t goes through L HGT layers to create the embedding vector $H^{(L)}[t]$. Such a mechanism ensures the final embedding vector of t is considered on multiple aspects through transformer architecture.

Optimization for Detection. For graph or node classification tasks, we use a multi-layer perceptron (MLP) with the softmax function as an activation function. The input of the layer depends on the prediction tasks. To reduce the effects of derivative saturation, we use cross entropy as the loss function

during training, and the parameters of our model are learned through back-propagation with gradient descent algorithms.

6.3.5 Two-Phase Vulnerability Detector

This component comprises two primary phases: *Coarse-Grained Detection* and *Fine-Grained Detection*. While the former phase assesses clean versus vulnerable smart contracts at the contract level based on the input bytecode or source code, the latter phase determines the line locations of vulnerabilities in the contract source code. One of our new contributions is detecting vulnerabilities at the line level, whereas earlier learning-based methods [76,77] only report vulnerabilities at the contract or function level.

Phase 1: Coarse-Grained Detection

This phase determines whether a smart contract is vulnerable. We employ the *heterogeneous contract graphs* and their embeddings for each input smart contract, and we train the MLP (Section 6.3.4) to predict whether a heterogeneous contract graph is clean or vulnerable. The MLP can produce a confidence score for each input graph with respect to each bug type; the contract is classified/predicted as buggy when the confidence score for the graph with respect to a bug type is greater than 0.5. This classification helps reduce the search space by filtering out likely clean smart contracts prior to the second phase of line-level vulnerability detection.

Phase 2: Fine-Grained Detection

For the smart contracts identified in the previous phase, the node embeddings of their *heterogeneous contract graphs* will go through node classification to determine if the nodes may be buggy. Similar to Phase 1, the MLP of the node classification step can produce a confidence score for each node in the input graph with respect to each bug type; a node is classified/predicted as buggy when the confidence score for the node with respect to a bug type is greater than 0.5. Note that the nodes correspond to statements or lines in source code, so we can identify the locations of vulnerabilities at the line level in source code [2].

6.4 Empirical Evaluation

We publicize the datasets and our graph embedding models at `https://github.com/MANDO-Project/ge-sc-transformer`.

[2]MANDO-HGT can also potentially detect bugs at the instruction level in bytecodes. However, when the bytecode instructions cannot be mapped back to source code lines, the detection results may not be readable by human developers. Thus, in this thesis, we do not perform instruction/line-level bug detection for bytecode.

Bug Types	**# Total / Buggy Contracts**	**# Total Nodes** (source/byte code)	**# Total Edges** (source/byte code)	**# Buggy Nodes** (source code only)
Access Control	114 / 57	13014 / 44475	10721 / 61896	7500
Arithmetic	120 / 60	17372 / 47967	14271 / 66020	10110
Denial of Service	92 / 46	13968 / 41066	11997 / 56711	8280
Front Running	88 / 44	22824 /51297	19761 / 71652	10008
Reentrancy	142 / 71	18898 / 46856	17614 / 64798	11238
Time Manipulation	100 / 50	16765 / 43424	15550 / 60464	10051
Unchecked Low Level Calls	190 / 95	17756 / 55103	14858 / 75950	7583
Total	846 / 423	120597 / 330188	119630 / 457491	64770

TABLE 6.1: Statistics of the Mixed Dataset.

Methods		**Access Control**	**Arithmetic**	**Denial of Service**	**Front Running**	**Reentrancy**	**Time Manipulation**	**Unchecked Low Level Calls**
Original Heterogeneous GNN	metapath2vec	48.43 72.80	54.89 69.52	64.13 69.46	66.79 69.86	64.28 62.38	60.73 64.88	62.74 69.23
Original Homogeneous GNNs	LINE	57.22 68.64	51.45 69.20	61.11 70.98	50.74 69.52	66.34 67.39	65.92 69.99	63.79 70.00
	node2vec	60.78 53.64	61.73 55.55	64.16 49.63	66.50 50.27	62.69 49.23	64.53 50.04	63.22 53.15
The best Buggy F1 scores of MANDO	Node features of the best scores	71.19 82.91	66.85 81.22	89.15 83.95	89.86 84.09	76.09 79.13	87.71 83.95	72.08 77.76
MANDO-HGT with Node Features Generated by	**NodeType One Hot Vectors**	82.86 79.46	88.13 **88.93**	89.33 85.93	92.69 **87.67**	93.84 **87.97**	95.68 81.39	80.11 76.14
	metapath2vec	**83.65** 78.56	88.86 86.93	88.91 86.69	93.70 86.15	**94.78** 87.05	95.89 81.89	**81.75** 77.07
	LINE	82.75 77.73	**89.41** 88.39	**89.89** 87.47	**95.23** 86.62	93.27 87.69	**95.99** 81.36	80.77 76.83
	node2vec	82.65 80.67	87.91 87.77	85.86 **87.63**	90.83 86.19	93.75 84.84	95.81 81.23	79.05 76.14

TABLE 6.2: Performance comparison in terms of Buggy-F1 score on different bug detection methods at the *contract* granularity level for both *source code* and *bytecode*. In each cell, the first number is the source code's result, and the second with grey shading is the bytecode's result. We use the *Heterogeneous Contract Graphs* of both clean and buggy smart contracts as the inputs for MANDO. The best performance for each bug type is in boldface separately for source code and bytecode. For MANDO, we report the best performance among node feature generators.

6.4.1 Dataset

Similar to the MANDO's experiments in Section 5.4, our evaluation is also carried out on a mixture of three datasets: **Smartbugs Curated** [73, 74], **SolidiFI Benchmark** [75], and **Clean Smart Contracts from Smartbugs Wild**, with the focus on seven bug types *Access Control*, *Arithmetic*, *Denial of Service*, *Front Running*, *Reentracy*, *Time Manipulation*, and *Unchecked Low Level Calls*. However, unlike MANDO's experiments, which only require the smart contracts in all the datasets to be in source code form, the bytecodes of the corresponding smart contracts are needed for the evaluation. Thus, we first employ Slither [123] to traverse and generate the basic homogeneous forms of CFGs and CGs for the input source code. To get smart contract bytecode, we use Cryptic compiler [223], a Python wrapper of the Solidity compiler, with the Solc versions flexibly depending on the declared versions in the source code, to generate the runtime bytecode from these source files. Then, we use

the EtherSolve tool [224] to build basic CFGs for bytecode. We have also developed a component for transforming the traditional CFGs and CGs generated by EtherSolve or Slither to our heterogeneous CFGs and heterogeneous CGs and then fuse them into *heterogeneous contract graphs* before extracting meta relations and feeding them to the **MANDO-HGT GNN** component.

Also, we randomly take some smart contracts from the clean set and mix them with the buggy set. We keep a ratio of 1:1 between clean and buggy contracts, in order to have more balanced training/test datasets for both graph and node classification tasks, following practices used in other deep learning-based bug detection studies in the literature that use more or less balanced datasets, e.g., SySeVR [145], Russell *et al.* [140]. Table 6.1 shows the actual numbers of buggy contracts and total numbers of contracts used in our experiments for each bug type, as well as the total numbers of nodes and edges in the constructed heterogeneous contract graphs for source code and bytecode. Note that for the fine-grained line-level bug detection task, our approach requires line-level labels for the bugs but some other datasets, such as the ones of Zhuang *et al.* [76], Liu *et al.* [77], and eThor [78] are not suitable for our experiments because they only have coarse-grained contract- or function-level labels for the bugs.

Note that the employed Slither and Ethersolve parsers do not automatically generate clean or vulnerable labels for a node. Instead, we wrote automated scripts to label the nodes based on the lines of vulnerable code identified either manually by Smartbugs authors or injected by the SolidiFI tool. For instance, Smartbugs authors label Line 11 in Figure 6.1 as containing a *reentrancy* bug; then our scripts labeled the nodes with red text in the heterogeneous CFG and CG as vulnerable.

6.4.2 Evaluation Metrics

Similar to Section 5.4, our prediction results are binary (clean versus vulnerable) classification of a node or graph, so we measure the prediction performance using the commonly used F1 scores. An F1-score evaluates the performance of a model's prediction by taking the harmonic mean of precision and recall of the model for a given class label. As detecting bugs is the main interest of our evaluation, we measure the F1-score metrics to evaluate the performance of the models when classifying vulnerabilities; for the bug label, and we refer to this metric as *Buggy-F1* [3].

6.4.3 Baselines and Parameter Settings

Baselines. To show the advantages of heterogeneous graph learning over homogeneous graph learning, we use both of them in our evaluation: We apply *metapath2vec* [41] as the heterogeneous graph neural networks, while

[3]To measure the effects of imbalances between clean and buggy data, Macro-F1 is also often considered in the literature by averaging the F1 scores of all class labels. However, in our case, we used a ratio of 1:1 to balance the amount of clean and vulnerable contracts and found that Macro-F1 scores are very close to Buggy-F1 and omitted them in the chapter.

Methods		Access Control	Arithmetic	Denial of Service	Front Running	Reentrancy	Time Manipulation	Unchecked Low Level Calls
Conventional Detection Tools	securify	13.0	0.0	18.0	53.0	23.0	24.0	11.0
	mythril	34.0	73.0	41.0	63.0	19.0	23.0	14.0
	slither	32.0	0.0	13.0	26.0	15.0	44.0	10.0
	manticore	30.0	30.0	12.0	7.0	9.0	24.0	4.0
	smartcheck	20.0	22.0	52.0	0.0	22.0	44.0	11.0
	oyente	21.0	71.0	48.0	0.0	20.0	24.0	8.0
Heterogeneous GNN	metapath2vec	35.46	68.70	60.64	80.65	71.66	67.51	26.06
Homogeneous GNNs	GCN	43.92	65.69	64.06	81.09	71.76	68.70	38.13
	LINE	53.59	68.61	62.28	83.06	74.78	70.76	7.10
	node2vec	44.94	67.84	63.92	81.84	71.52	67.81	34.26
The best buggy scores of MANDO	Node features of the best scores	81.98	84.35	82.12	90.51	86.40	90.29	84.81
MANDO-HGT with Node Features Generated by	**NodeType One Hot Vectors**	**89.46**	**91.18**	**86.81**	**94.02**	**92.59**	**95.04**	90.09
	metapath2vec	78.99	82.99	76.54	89.13	84.06	89.83	76.05
	GCN	87.76	88.86	83.73	92.96	89.59	92.91	89.47
	LINE	86.58	87.98	83.00	92.17	88.77	93.26	**90.89**
	node2vec	84.23	84.66	81.93	90.46	88.48	92.31	87.60

TABLE 6.3: Performance comparison in terms of Buggy-F1 score on different bug detection methods based on *source code* at the *line* granularity level. The best performance for each bug type is in boldface.

applying *node2vec* [6], *LINE* [7], and *GCN* [8] as homogeneous GNNs. Node embeddings generated from the baseline GNNs are used as the **Node Feature Extractor** for input node features of our **MANDO-HGT GNN** and the prediction baselines as well. Additionally, we use some variants of MANDO in Chapter 5, a framework specialized in smart contract vulnerability detection based on GAT [9] and HAN [11] graph neural networks with multiple dynamic customized metapaths, as the baselines. We also compare our line-level source code bug detection method to six widely used smart contract vulnerability detection tools based on conventional software analysis techniques: *Manticore* [117], *Mythril* [214], *Oyente* [127], *Securify* [126], *Slither* [123], and *Smartcheck* [128].

Parameter Settings. All models have their node or graph embedding size set to 128. We employ an adaptive learning rate ranging from 0.0005 to 0.01 for coarse-grained classification and from 0.0002 to 0.005 for fine-grained classification. For each target node and its each meta-relation pair that are fed to the **MANDO-HGT GNN**, we use two HGT layers [12] and set eight multi-heads whose hidden size is 128. We use their recommended settings for node2vec, LINE, GCN, metapath2vec, and MANDO to ensure the highest performance.

6.4.4 Experimental Results

In our initial experiments, we divided the 423 buggy contracts and 2,742 clean contracts into the training/validation/test sets using the 60%/20%/20% split ratio. However, some bug types in our dataset have fewer than 50 contracts, resulting in insufficient training/testing samples. In addition, we discovered that the loss value remains constant after a fixed number of epochs (100 and 50 epochs for Fine-Grained and Coarse-Grained tasks, respectively). In order to increase the training/test set sizes and maintain the proportion of vulnerable

nodes in each set, we decided to split the contracts into only training/test sets using the 70%/30% ratio for all the bug types. For each setting, embedding method, and bug type, to ensure the robustness of our results we repeated the experiments 20 times with a different random seed and performed t-tests, and report the average results.

Coarse-Grained Contract-Level Vulnerability Detection

Table 6.2 shows the average results of 20 independent runs of the baselines and MANDO-HGT at the contract level for both source code and bytecode. We can observe that:

- MANDO-HGT and MANDO with node type generated by one hot vector [21] perform better than other methods, i.e., original heterogeneous and homogeneous GNNs for detecting bugs at source code or byte code levels. For instance, MANDO-HGT improves up to 34.52% compared to heterogeneous GNN for detecting *arithmetic* bugs with source code inputs. Moreover, no matter what methods are used to generate node features, MANDO and MANDO-HGT frameworks still outperforms the baselines.

- Between MANDO-HGT and MANDO, the performance of MANDO-HGT is overall higher than that of the former for both source code and bytecode. However, the performance gaps between the two frameworks for some bug types are relatively small, e.g., 2.24% in access control bugs in the bytecode. For this reason, we apply a t-test to check if the performance of MANDO-HGT is statistically significantly better than MANDO based on the 20 runs of both models for each bug type. All of the t-values are positive while most p-values are less than 0.05, which implies that the performance of MANDO-HGT for most bug types is statistically significantly better than MANDO.

- It is clear that integrating node features via different graph neural networks inside MANDO-HGT outperforms all other GNN baselines, although it is unclear which node features perform the best. This observation is also applicable to various node features used by previous studies, e.g., MANDO. Hence, we believe that an architecture combining different GNNs is useful for classifying buggy contracts.

Fine-Grained Line-Level Vulnerability Detection

Table 6.3 shows the performances of MANDO-HGT and other baselines based on source code at the *line level*. From the results, we observe that:

- MANDO-HGT generally outperforms conventional analysis-based bug detection tools and basic GNNs. The performance improvements are around 10.96%–76.89% in Buggy-F1 scores, for different bug types. For example, for the *time manipulation* bugs, we got a 95.04% Buggy-F1

score, considerably higher than the best 70.76% among the baseline conventional tools and basic GNNs. Some conventional detection tools in Table 6.3 hardly function (Buggy-F1=0%) for certain vulnerability types due to their inherent limitations in relying on predefined patterns that are incapable of capturing these vulnerabilities.

- In MANDO-HGT, node features generated by one hot vector are better than other node feature generating methods for most bug types. The only exception is the node features generated by LINE in MANDO-HGT for detecting *unchecked low level calls* bugs. For example, for the *reentrancy* bugs, we got the best Buggy-F1, 92.59%, with the node features based on NodeType One Hot vectors. The models with node features generated by other methods are all lower.

6.4.5 Case Studies: Interpreting Vulnerability Prediction Results

To shed light on the reasons why MANDO-HGT can produce successful or failed predictions, we aim to identify certain correlations between MANDO-HGT's prediction results and the actual semantics of smart contract code in this section. We use a post-hoc local model-agnostic interpretable framework, GraphSVX [221], a state-of-the-art method for graph interpretability, based on Shapley value [225] for graph neural networks to interpret the behaviors and confidence of our model's predictions versus the input smart contract source code. We also evaluated several other XAI methods, including the original SHAP, and found that the results of GraphSVX were better than others since GraphSVX considers more of the structures of graphs while the SHAP only considers individual object embeddings. Shapley values of an individual feature j is $\phi(val(j))$ providing a proxy to assess the overall significance of the feature j to an output of a data point by averaging the marginal contribution of the feature across all possible coalitions where the feature presents. In MANDO-HGT, when making a prediction on a *focal* node, we consider all neighbors N of this node as the features which could impact the prediction on the *focal* node. We obtain the Shapley value by:

$$\phi(val(j)) = \sum_{S \subseteq \{1,\ldots,N\}\setminus\{j\}} \frac{|S|! \times (N-|S|-1)!}{N!} (val(S \cup \{j\}) - val(S)),$$

where $S \subseteq \{1,\ldots,N\}\setminus\{j\}$ are the possible coalitions of node j's neighbors, and $val(S) = \mathbb{E}\left[f(\mathbf{X}) \mid \mathbf{X}_S = \mathbf{x}_s\right] - \mathbb{E}[f(\mathbf{X})]$ with $\mathbb{E}[f(\mathbf{X})]$ being average prediction of dataset $\mathbf{X}$.

We extract neighbor nodes of a focal node and compute their marginal contribution towards the prediction for the focal node, and use some random masking strategy [221] to select subsets of them to calculate their Shapley values more efficiently. The Shapley value of a node implies the contribution of the node to the focal node, and a node's confidence refers to the confidence score of the model's prediction for the node. A node is considered buggy if

its confidence score is greater than 0.5 with respect to a bug type (cf. Section 6.3.5).

For any given node used as a focal node for examination, we calculate all of its neighbors' Shapley values to find out their effect on the model's bug prediction result. Larger Shapley values indicate the neighbor's higher impact on the focal node. In the following, we explain several cases where MANDO-HGT makes correct or incorrect predictions to further study possible correlations between the Shapley values of a focal node's neighbors and its meta relations. For instance, a buggy node might belong to several meta relations, and if there is a meta relation that frequently contains nodes of a certain bug type and contributes the highest Shapley value to the focal node, the focal node's bug prediction confidence might be high. On the contrary, if some meta relations contain clean nodes contributing high Shapley values to the focal node, the bug prediction confidence for the focal node might be low.

From Figure 6.5 to Figure 6.8, subgraphs of the heterogeneous contract graphs are shown next to their source code. The nodes are filled with either red or green, indicating either buggy or clean predictions by our model for the nodes. The circle or triangle shapes of the nodes stand for buggy or clean nodes as indicated by the ground-truth labels. The yellow tag for each node contains information on the node type, its corresponding lines of source code, the confidence score of our model prediction (when the confidence is higher than 0.5, the node is predicted as buggy; otherwise, clean), and the Shapley value of that node to the focal node at the end of the arrow chain in each subgraph.

True positive cases

Figure 6.5-(1) shows a code snippet together with a sub-graph of its heterogeneous contract graph that contains an *access control* bug at Line 27 corresponding to the focal node EXPRESSION. We saw a meta-relation pair $\langle FUNCTION\_NAME, next, ENTRY\_POINT\rangle$ and $\langle ENTRY\_POINT, next, EXPRESSION\rangle$ frequently appear in our samples. In this case, $FUNCTION\_NAME$ and $ENTRY\_POINT$ nodes correspond to the whole `PopBonusCode` function were predicted as a buggy node and had approximate Shapley values 0.506 and 0.494 contributing to the focal $EXPRESSION$ node, which implies that the focal node is likely buggy too. Indeed, MANDO-HGT correctly predicted the focal $EXPRESSION$ node as buggy.

Figure 6.5-(2) illustrates a smart contract that has arithmetic bugs at Lines 15, 18, 21, 24, 27, and 30 because the `input` variables in these functions are not checked for overflow or underflow before the operations are performed. One of the meta-relation pairs, $\langle FUNCTION\_NAME, next, ENTRY\_POINT\rangle$ and $\langle ENTRY\_POINT, next, EXPRESSION\rangle$, appears three times in Lines 15, 18, and 21. Lines 24, 27, and 30 correspond to another meta-relation pair: $\langle FUNCTION\_NAME, next, ENTRY\_POINT\rangle$ and $\langle ENTRY\_POINT, next, NEW\_VARIABLE\rangle$. The $FUNCTION\_NAME$ and

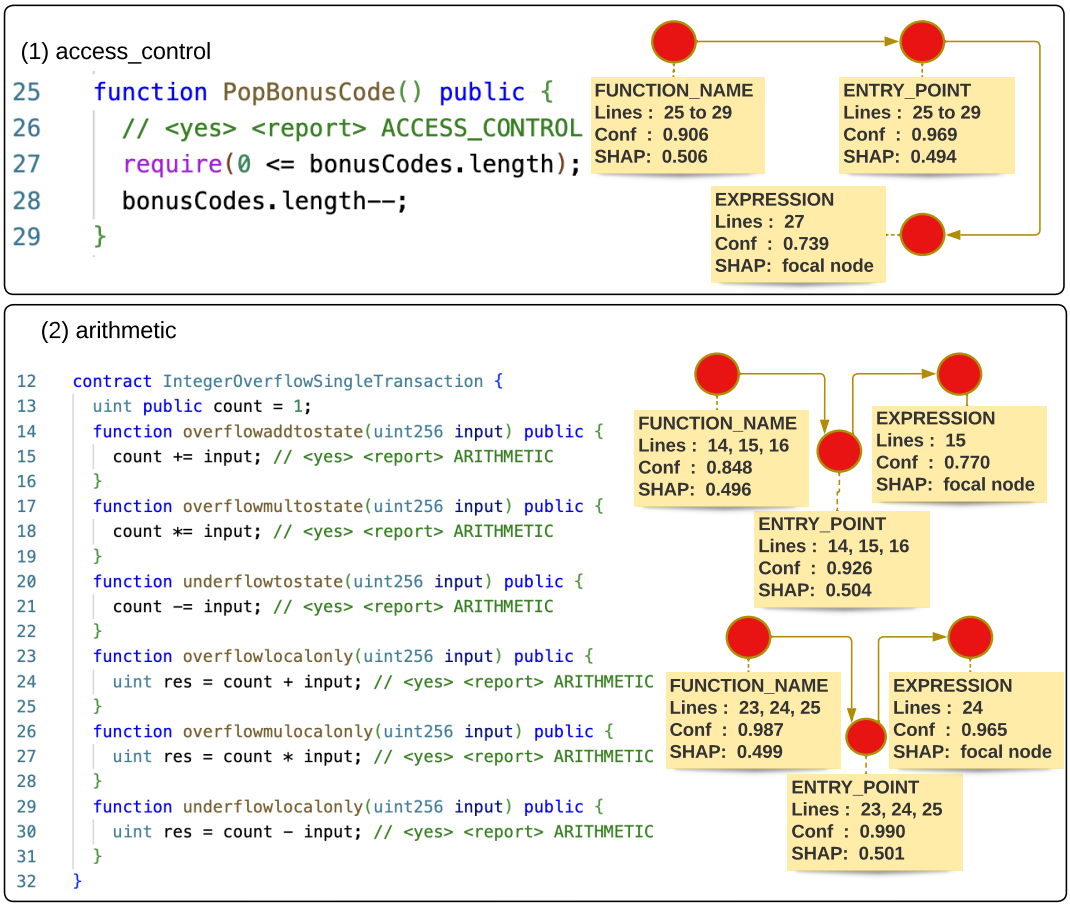


FIGURE 6.5: True positive cases of *access control* and *arithmetic* samples.

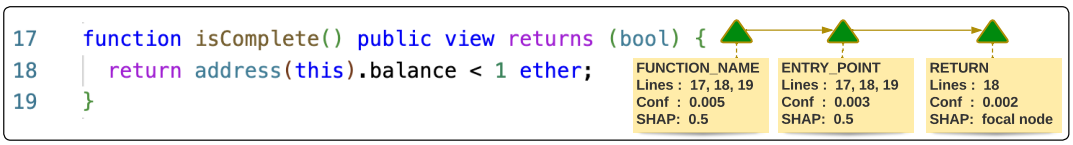


FIGURE 6.6: True negative case of *access control* sample.

ENTRY_POINT nodes represent an entire function and its entry point, indicating such bugs often happen at the beginning of a function with specific operations. MANDO-HGT is good at recognizing such frequently appearing bug patterns. Correspondingly, the Shapley values of these nodes to the focal buggy *EXPRESSION* and *NEW_VARIABLE* nodes are around 0.5, which implies that relatively high Shapley values might reflect frequently occurring patterns for bug detection.

True negative cases

The code in Figure 6.6 generated a heterogeneous contract graph with meta-relation pair
$\langle FUNCTION\_NAME, next, ENTRY\_POINT \rangle$ and
$\langle ENTRY\_POINT, next, RETURN \rangle$. Its meta relations with *EXPRESSION*,

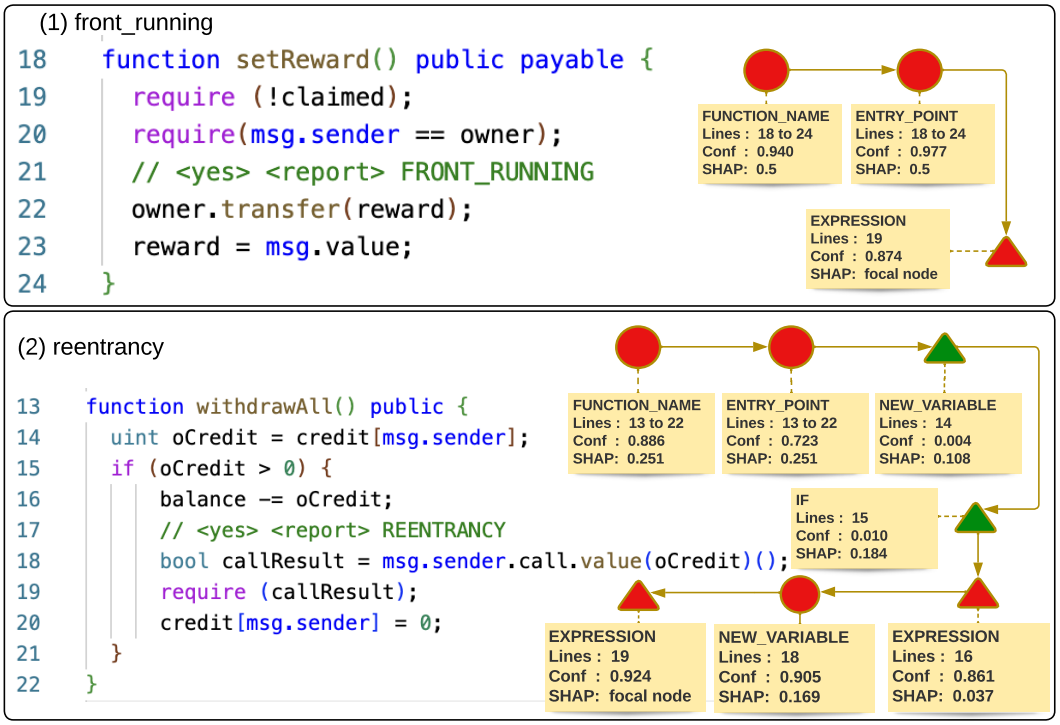


FIGURE 6.7: False positive cases of *front running* and *reentrancy* samples.

ENTRY_POINT, and *FUNCTION_NAME* node types also *differ* from those of *access control* bugs (e.g., the one in Figure 6.5-(1)). Besides, the Shapley values of *ENTRY_POINT* and *FUNCTION_NAME* nodes of function `isComplete` to the focal *RETURN* statement, in this case, are 0.5, indicating that the two nodes (which are clean) have relatively significant impact on the prediction for *RETURN*. MANDO-HGT correctly predicted the focal node as a clean node (via a very low bug confidence score 0.002).

False positive cases

MANDO-HGT may wrongly predict some clean code as buggy. For example, for *front running* vulnerabilities, which are typically found in a statement that attempts to transfer a high amount for their transaction to be prioritized, they often involve an *EXPRESSION* node, similar to the node for the line 22 in Figure 6.7-(1). However, some clean statements occasionally preceding the buggy line are also *EXPRESSION* nodes, such as lines 19 and 20 in function `setReward`. The Shapley values from the buggy *FUNCTION_NAME* and *ENTRY_POINT* nodes are 0.5 and 0.5, which indicated that they have significant impact to that focal node, causing our model to mistakenly classify the focal node at Line 19 as buggy.

Figure 6.7-(2) illustrates a similar false positive situation for the reentrancy bug type. The *NEW_VARIABLE* node corresponds to the code Line 18 defining a new variable and has a reentrancy bug in relation to the reduction of relevant `credit` to zero at Line 20. The focal *EXPRESSION* node at Line

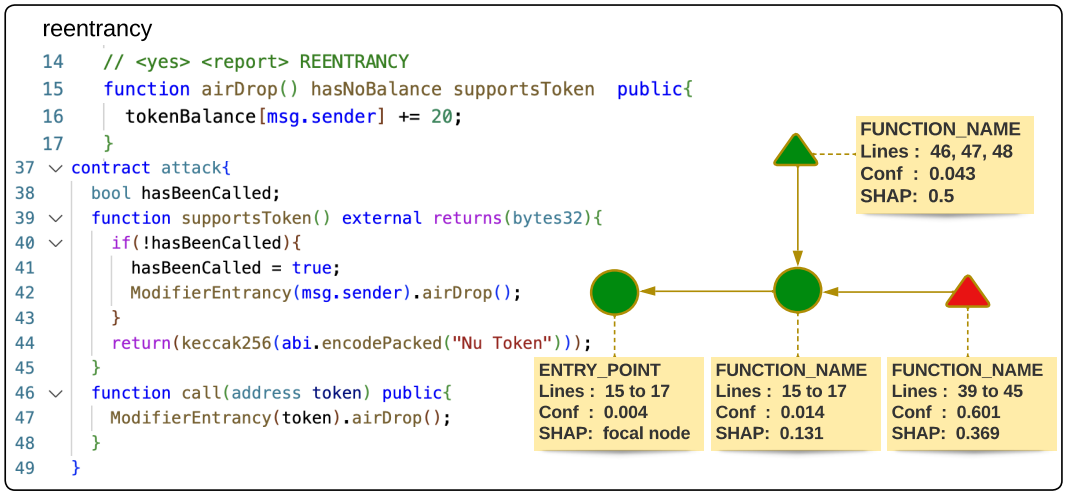


FIGURE 6.8: False negative cases of *arithmetic* and *reentrancy* samples.

19 was wrongly predicted by MANDO-HGT, likely because it has data dependency with `callResult` defined at the buggy line 18. The buggy nodes like *FUNCTION_NAME*, *ENTRY_POINT*, and *NEW_VARIABLE* at Line 18 have relatively higher Shapley values of 0.251, 0.251, and 0.169 to the focal node at Line 19 than the Shapley values from the other two clean nodes (*IF* at Line 15 and *NEW_VARIABLE* at Line 14), 0.184 and 0.108. This indicates that the buggy nodes have more impact on the focal *EXPRESSION* node at Line 19, causing MANDO-HGT to misclassify it as buggy.

False negative cases

MANDO-HGT may also miss certain vulnerabilities. For example, Figure 6.8 shows a *reentrancy* bug in the focal node *ENTRY_POINT* involving lines 15,16,17, but our model predicts it as a clean node. By considering the meta relations pair $\langle FUNCTION\_NAME, next, FUNCTION\_NAME \rangle$ and $\langle FUNCTION\_NAME, next, ENTRY\_POINT \rangle$ of the focal node, we have the *clean FUNCTION_NAME* node containing lines 46, 47, 48 with 0.5 Shapley value having the greatest impact to the focal *ENTRY_POINT* node, higher than the 0.131 Shapley value from the buggy *FUNCTION_NAME* node containing lines 15,16,17, causing our model to predict the focal node as clean.

6.4.6 Limitations and Discussions

We know that the effectiveness of heterogeneous graph transformers relies on node/edge types and meta relations used, besides various hyperparameters for training. The graphs used to represent syntactical and semantic information from smart contract source code or bytecode also significantly impact graph learning. Especially for bytecode, whose syntactical structure is flatter with more obscured semantic information than source code, suitable graph representations become more important for effective learning. Also, although our model can detect fine-grained bugs at the instruction level for bytecode as

well, it would be better to map the detected bugs in bytecode back to source code lines for better readability when reporting the results to developers. We restrain our tool from reporting instruction-level bug detection results before we have a reliable way to make the results understandable in relation to their source code.

Lacking labeled data is always an issue when applying supervised learning to classification tasks, especially for smart contracts with limited sample buggy code for training. Although Smartbugs and SolidiFI datasets are useful, some smart contracts in the datasets were not annotated. Furthermore, the labels annotated in the datasets for some bugs may not always be objective and agreeable by all developers as there can be subjective and more than one interpretation of the root causes of a bug (e.g., bugs due to some missing lines of code); some labels are not fine-grained enough for each line or even each expression in code. Such inconsistent labels may hinder the training of our models. In our experiments, we excluded unlabelled data and manually checked some data samples and corrected a few inconsistent labels, and used balanced data sets for training and testing, to minimize the impact of inaccurate labels or imbalanced data.

In the experiments, some smart contracts could not be processed by Slither [123] or EtherSolve [222]. One of the main limitations of the tools is that they rely on the Solidity compiler to build the control-flow graph of the contracts, but there are issues related to the version compatibility of Solidity or the uses of optimized/obfuscated bytecode. For example, a smart contract written in an older version of Solidity is not supported by the compiler used by Slither or EtherSolve; the control-flow graph may not be built successfully.

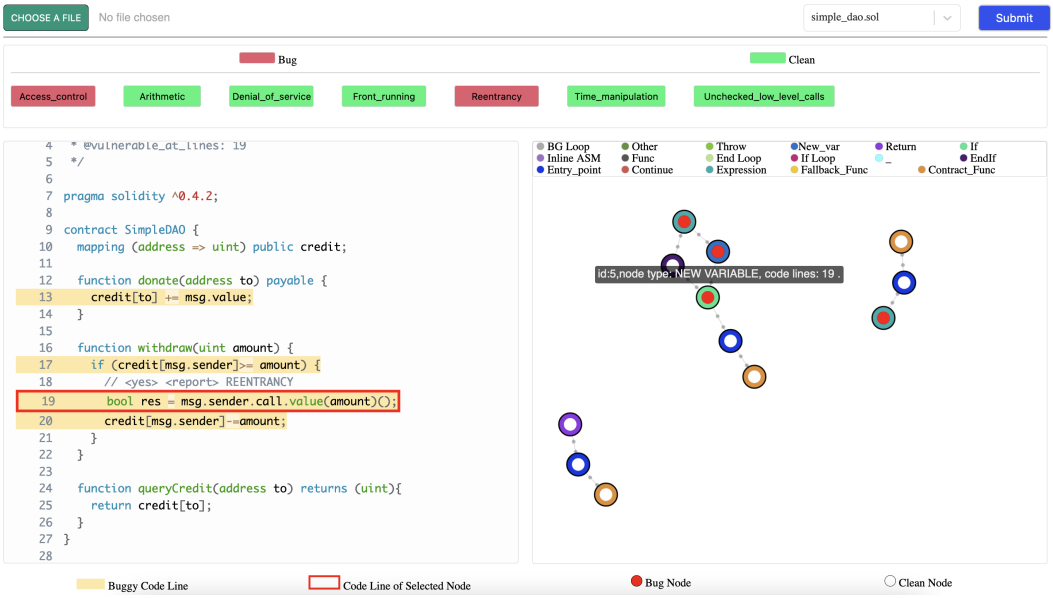
Chapter 7

A Tool for Vulnerability Detection for Smart Contract Source Code by Heterogeneous Graph Embeddings

Build on the MANDO framework in Chapter 5, we propose a new graph learning-based tool, MANDO-GURU, that aims to accurately detect vulnerabilities in smart contracts at both coarse-grained contract-level and fine-grained line-level. Using a combination of control-flow graphs and call graphs of Solidity code, we leverage the heterogeneous graph attention neural networks presented in Section 5.3.4 as core backend components to encode more structural and potentially semantic relations among different types of nodes and edges of such graphs and use the encoded embeddings of the graphs and nodes to detect vulnerabilities. Our validation of real-world smart contract datasets shows that MANDO-GURU can significantly improve many other vulnerability detection techniques by up to 24% in terms of the F1-score at the contract level, depending on vulnerability types. It is the first learning-based tool for Ethereum smart contracts that identifies vulnerabilities at the line level and significantly improves the traditional code analysis-based techniques by up to 63.4%.

7.1 Introduction

Leveraging on the built technical architecture of the MANDO framework presented in Chapter 5, in this chapter, we propose a new tool with a new method for representing smart contracts as specialized graphs and learning their patterns automatically via graph neural networks on a large scale to detect vulnerabilities at both line-level and contract-level accuracy. In particular, (1) we represent Ethereum smart contract source code written in Solidity as *heterogeneous contract graphs* that combine control-flow graphs (CFGs) and call graphs (CGs) using unique properties of Solidity to capture contract code semantics, and (2) we design specialized metapaths for the graphs and build heterogeneous attention graph neural networks to learn multi-level embeddings of the contract code in various levels of granularity, which are then used together with known instances of smart contract vulnerabilities to train classifiers that can recognize vulnerabilities accurately in new smart contract code at both line-level and contract-level. Our tool is named MANDO-GURU. We have constructed a dataset containing both buggy and clean smart contracts, and compared MANDO-GURU with several state-of-the-art and conventional baselines. Our validation results show that MANDO-GURU outperforms the baselines in both contract- and line-level vulnerability detection with significant improvements. Our tool is publicly available at `https://github.com/MANDO-Project/ge-sc-machine`. A test version is currently deployed at `http://mandoguru.com`, and a demo video of our tool is available at `http://mandoguru.com/demo-video`.




```
4  * @vulnerable_at_lines: 19
5  */
6
7  pragma solidity ^0.4.2;
8
9  contract SimpleDAO {
10   mapping (address => uint) public credit;
11
12   function donate(address to) payable {
13     credit[to] += msg.value;
14   }
15
16   function withdraw(uint amount) {
17     if (credit[msg.sender]>= amount) {
18       // <yes> <report> REENTRANCY
19       bool res = msg.sender.call.value(amount)();
20       credit[msg.sender]-=amount;
21     }
22   }
23
24   function queryCredit(address to) returns (uint){
25     return credit[to];
26   }
27 }
28
```

FIGURE 7.1: A sample vulnerability detection page of MANDO-GURU for an Ethereum smart contract includes summary detection results for seven bug types (Top), code snippet (Left), and its corresponding heterogeneous contract graph (Right). Line 19 with a yellow background is the root cause of a Reentrancy bug (Left); the nodes containing the Reentrancy bug are highlighted with red (Right).

7.2 Usage

Figure 7.1 illustrates MANDO-GURU's main user interface and core features. More specifically, after a user submits a Solidity source file using the submit button on the top, MANDO-GURU scans the input and summarizes the coarse-grained contract-level detection results of seven bug types (the red/green buttons near top). A red button indicates a bug type detected for the contract, and users can click it to show the fine-grained line-level detection results. On the left side of the figure, the source code lines containing detected bugs would be highlighted with a yellow background. The right side visualizes the corresponding heterogeneous contract graph of the input contract. If a node is detected having a bug, it is colored red. When users hover the pointer over a node, the node details will be shown, and when they click a node, the code lines relevant to that node will be marked with the red border on the left.

Besides the core features, MANDO-GURU also provides various statistics charts for general analyses of the generated heterogeneous contract graphs. In particular, after getting the detection results, users could click the "Show Statistics" button to get three extended charts, including the number of clean and buggy nodes, running time for coarse-grained and fine-grained detection, and the density of each bug type. We explain in detail the charts in our demo video.

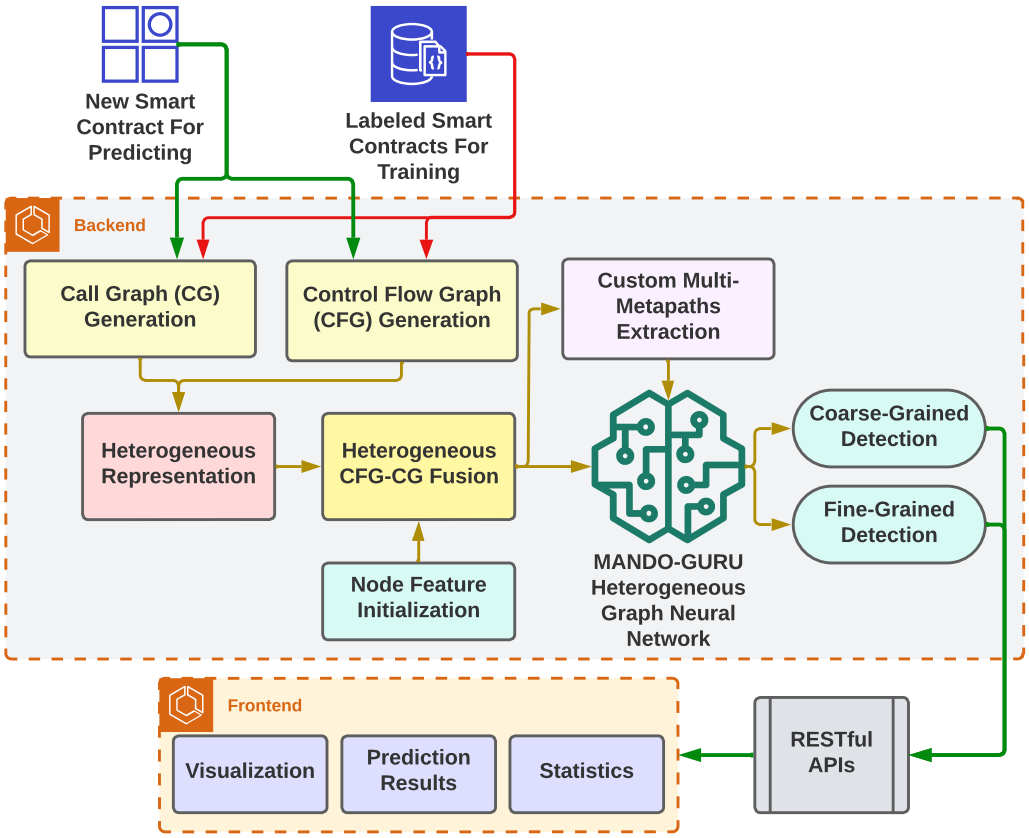


FIGURE 7.2: Overview of the MANDO-GURU Tool.

7.3 Tool Design & Implementation

Figure 7.2 illustrates an overview of MANDO-GURU with three main components: *Backend*, *RESTful APIs*, and *Frontend*. Backend plays a vital role with several core sub-components such as heterogeneous representation for the generated graphs from input smart contracts, heterogeneous graph fusion, custom multi-metapaths extraction, heterogeneous graph neural network, and vulnerability detections in coarse-grained and fine-grained levels. The technical details of *Backend* are described in Section 5.3. The Frontend component services are used to visualize the prediction results and the statistics of the analyzed smart contracts. RESTful APIs are implemented as a bridge to communicate between the Backend and the Frontend.

7.3.1 Backend

Heterogeneous Representation for the Generated Control-Flow Graphs and Call Graphs

First, to generate the basic control-flow graphs and call graphs, we use Slither [123] to process the source code of each input Ethereum smart contract. Then, we convert the graphs into heterogeneous forms, called *heterogeneous control-flow graphs (HCFGs)* and *heterogeneous call graphs (HCGs)*, to represent

the relations of different node and edge *types* and graph topologies. In particular, a heterogeneous graph is defined as a special graph consisting of multiple-type of nodes or edges. Unlike some recent studies [76, 163] that use only homogeneous graph structures and lead to loss of valuable information on the code semantics in smart contracts, one primary contribution of MANDO-GURU is to focus on capturing and retaining more structures and semantics of source code through our heterogeneous representations.

Fusion of Heterogeous Control-Flow Graphs and Heterogeous Call Graphs

An HCFG can represent each function in a smart contract, and it contains an entry node corresponding to the entry point/header of the function. Generally, a smart contract may be considered as a set of HCFGs since it consists of more than one function. The invocation relations among the functions in one contract or between contracts are represented by HCGs.

The structures of the heterogeneous graphs can be shared or combined to enrich information for graph learning. Hence, we design a sub-component as a core fusion of HCGs and HCFGs into a global graph. Accordingly, the HCG edges of a contract act as bridges to link the discrete HCFGs of the contract functions into a global fused graph. We call the fusion graphs as *heterogeneous contract graphs*. Intuitively, for each and every function node i in the call graph G_C, the function control-flow graph G^i_{CF} is attached to the function node i at the entry node of G^i_{CF}, and thus the call graph G_C is expanded with control-flow graphs to produce the heterogeneous contract graph G_{Fusion}. The heterogeneous graph generation also allows us to expand the generalizability of the proposed method to other programming languages (e.g., C/C++, Java) with minor modifications.

Node Feature Initialization

In the default setting of MANDO-GURU, the one-hot vectors based on node types are used to initialize node features. Besides, various state-of-the-art node embedding techniques can be plugged into MANDO-GURU to capture the graph topology and extract the node features. For a more comprehensive validation of the effectiveness of various initialization of node features, we use both embedding methods for homogeneous graphs (e.g., node2vec [6]) and embedding methods for heterogeneous graphs (e.g., metapath2vec [41]) (see Section 7.4).

Extraction of Custom Multi-Metapaths

Build on MANDO framework, MANDO-GURU tool also extracts length-2 metapaths of each node type pair from a heterogeneous contract graph since learning the extracted metapaths can be an effective way to learn the graph structures [11, 41]. Similar to the method used in HAN [11], we only focus on metapaths of length 2 to capture the relations between each node type pair and its neighbors and to prevent the explosion of metapaths when the

generated heterogeneous contract graphs contain a dynamic number of node types (reaching eighteen in some large smart contracts, with five different connections per node type) and pre-defining all possible metapaths with any length according to all possible node types and edge types would lead to an exponential explosion of metapaths, increased data sparsity, and reduced accuracy in training data.

In addition, the heterogeneous contract graphs have mostly tree-like structures, with very few of their own back-edges induced by the LOOP-related statements in the smart contracts' source code, leading to the lack of metapaths connecting many types of leaf-node in the graphs. Therefore, we customize the length-2 metapaths by reflecting the relation R_i between adjacent nodes, from type A_i to type A_{i+1} and also from A_{i+1} to A_i to extract multiple-metapaths.

Heterogeneous Graph Neural Network

Our unique heterogeneous graph neural network learns to weigh the importance of every metapath and node by the node-level attention mechanism and can handle multiple dynamic custom metapaths without pre-defining the list of input metapaths. In particular, with the initialized node features (node embeddings) $e_i^{\phi_k}$ for each node i whose type is ϕ_k; then, we construct a corresponding weighted node feature by a linear transformation. Next, we measure the weight of the t-th metapath according to the node type ϕ_k of (i, j) pair by leveraging the self-attention mechanism [212] between i and j.

We concatenate all node embedding $M_i^{\phi_k}$ corresponding to all node type ϕ_k of all node i to generate a unified embedding vector for a node, which is used to train a *fine-grained* bug classifier. The average of all node embeddings in a graph is used as the graph embedding, which is used to train a *coarse-grained* bug classifier. Also, we employ the multi-layer perceptron (MLP) with a softmax activation function for predicting, with the inputs depending on the type of detection tasks. Moreover, the loss function for the training process is cross-entropy, and the parameters of our model are learned through back-propagation.

Coarse-Grained Detection and Fine-Grained Detection

First, MANDO-GURU classifies if a contract is clean or contains a type of vulnerabilities at the contract level by using coarse-grained graph classification. Next, MANDO-GURU identifies the actual locations of the vulnerabilities in the smart contract source code at the line level using fine-grained node classification. Providing line-level locations of vulnerabilities is one of our primary contributions, while the previous graph learning-based methods (e.g., [76,77]) only report vulnerabilities at the contract or function level.

7.3.2 RESTful APIs and Frontend

MANDO-GURU is based on the FastAPI framework [226] to create our RESTful APIs as well as validation data to handle the requests and respond

the detection results to the Frontend services. Also, we use a token for each request to validate and reduce the unexpected demands to our system via the basic HTTP authentication method. All RESTful APIs in MANDO-GURU are implemented and provided under POST methods. Besides, to ensure the MANDO-GURU's overall performance, we encode the source code of smart contracts to Base64 format before processing. Our APIs could be categorized into two groups depending on the request purposes from the Frontend component services: (1) Coarse-Grained requests for predicting whether a source code has any bug; and (2) Fine-Grained requests for getting the lines and nodes detected as having bugs.

Our Frontend web application is built on ReactJS [227] and ApexChartsJS [228] libraries. When users submit a source file to our web app, it scans through the file for a total of seven bug kinds supported and returns the summary and details of detection results for each bug type. We also provide some sample smart contracts in a dropdown menu, which may help the users who lack the Solidity source files to test MANDO-GURU more flexibly. The detection results are then visualized by interactive graphs and highlighted code snippets for users to double-check them easier.

7.4 Tool Validation

7.4.1 Setup

Based on the MANDO's experiments in Section 5.4, our evaluation uses two tasks: (i) contract-level vulnerability detection; and (ii) line-level vulnerability detection. We combine the three following datasets for our training: **(1) Smartbugs Curated** [73,74] **(2) SolidiFI Benchmark** [75] and **(3) Clean Smart Contracts from Smartbugs Wild** [73,74]. In total, we have 2,742 clean contracts and 493 annotated buggy contracts.

We use the following four state-of-the-art methods presented in Section 2.1.2 as the graph-based neural network comparison methods: *node2vec* [6]; *LINE* [7]; *Graph Convolutional Network (GCN)* [8]; and *metapath2vec* [41]. We use the output embeddings of the homogeneous and heterogeneous graph neural networks in two ways in our validation: First, directly as the baselines for the coarse-grained graph classification tasks and fine-grained node classification tasks. Second, each of the graph neural networks is plugged into MANDO-GURU as the topological graph neural network; the generated embeddings are considered the node features besides those based on the node-type one-hot vectors of the default setting and then fed to MANDO-GURU Heterogeneous Graph Neural Network (HGNN). We also used six detection tools built upon traditional software engineering techniques: *Manticore* [117]; *Mythril* [121]; *Oyente* [127]; *Securify* [126]; *Slither* [123]; and *Smartcheck* [128].

We use F1-score and Macro-F1 scores to measure the performance of our node/graph classification for the detection tasks. F1-score is used to validate the models' performance when finding bugs, and is also called *Buggy-F1*. Macro-F1 is considered to avoid biases in the clean and bug labels.

7.4.2 Empirical Results

We only briefly report highlights of our contributions and achievements here (See Section 5.4 for more comprehensive evaluations).

Contract-Level Vulnerability Detection

- MANDO-GURU outperforms baseline GNNs. E.g., an improvement of 24% in both metrics is achieved by MANDO-GURU over the best baselines for detecting the Front Running type of bugs.
- The node feature generatation methods help MANDO-GURU outperform all the baselines; and it shows that our architecture is general for plugging in various kinds of GNNs.
- Being competible with Slither [123] makes MANDO-GURU more effectively with various versions of Solidity; MANDO-GURU is able to find newly-appeared bugs that graph learning methods [76, 163] struggle to achieve.

Line-Level Vulnerabilty Detection

- MANDO-GURU outperforms conventional tools significantly with improvement up to 63.4% compared to the best performing tools for the Reentrancy type of bugs. It can be explained by (i) more CFG structures retained by our heterogeneous graphs; and (ii) the flexibility of our architecture.
- Our method beats the results of the baseline GNNs where the macro-F1 scores of our model is up to 20% higher than the ones of the baseline GNNs.
- Conventional detection tools perform well in detecting arithmetic bugs because they mostly use symbolic execution and such technique is suitable for detecting arithmetic bugs [215]. However, MANDO-GURU performance is still on par with the tools.

Chapter 8

Conclusion and Future Work

8.1 Summary of Contributions

With the expanding presence of social networks and decentralized systems like blockchain, the need for enhanced security analytics and investigations on these platforms is growing. Notably, these systems often utilize or can be represented as graph-structured data, requiring appropriate approaches to manage and analyze these data types. Based on graph representation learning approaches, this thesis addressed two vital challenges regarding security analytics in this context: (1) Utilizing social network analysis to aid criminal investigations and (2) Detecting vulnerabilities in blockchain smart contracts through heterogeneous graph embeddings.

8.2 Utilizing social network analysis to aid criminal investigations

In the early stage, we presented SoChainDB, a framework to crawl blockchain-based social networks. Its robust and general architecture can handle various kinds of blockchain systems. Along with our system, we provide the public dataset of Hive - one of the largest blockchain social networks. We also discussed and released the data of Splinterlands, a collectible decentralized card game, and NFTShowroom, a platform for purchasing the ownership of digital arts, both built upon Hive blockchain technology. With over 100 GB of post-processed data, SoChainDB allows for large-scale combined analysis of social networks' various aspects, especially in security topics. All data presented in this thesis is ready and accessible via our website through a RESTful API service or archival data files. Several research directions on information retrieval might fit this blockchain-powered social network database:

- *Massive Scale Social Network Analytics:* Based on over 100 GB of post-processed data, besides common social networking factors, extensive research can exploit the unique blockchain characteristics, such as users' motivation to contribute highly-rated content through the built-in crypto-tokens and rewards mechanisms, to explore the impact of articles. Hence, the assessments could be more comprehensive than similar approaches on the regular social network data.

- *Cross-domain Behavior Analytics:* Since we can obtain the entire Hive data, we could use the data to answer different basic questions related to the behavior of users across services. For example, game players could use social network platforms to publicize their achievements and build friendships with other players. Such activities allow us to understand users' behaviors in various domains and build a more accurate recommender system that offers helpful information.

- *Impact of Reward on User Engagement:* The reward system is a unique feature of blockchain-based social networks. Understanding the causality between earning and user activities in social networks can open a

direction to redesign existing social networks toward offering a better user experience.

In the deep stage of social network analysis, we addressed the problem of utilizing link prediction to aid criminal investigations by proposing two network-based learning frameworks to automatically predict the links among offenders or crime cases. The first framework is a simple similarity-based unsupervised approach that does not require any labeled training instances, as it makes predictions solely based on the network topology. The second framework, DEAL, is a supervised approach that efficiently aligns the network topology with its node attributes to predict links, specifically for unseen newly emerged nodes. Since the unsupervised approach does not require training phases, it is better in scalability and has less need for memory requirements. However, it only works for transductive link prediction, which is suitable for predicting upcoming crimes based on the previous network structure. Alternatively, inductive link prediction is beneficial for finding connections between a new unknown case and existing ones.

Although experimental results indicate that the proposed frameworks are reliable and improve state of the art in criminology, several enhancements can be made in the future:

- The proposed frameworks, especially the supervised DEAL framework, deal with an offline setting. As a future direction, we will adapt the DEAL framework to the online setting, where the model is updated upon receipt of new instances.

- As mentioned in Section 4.2, we build on a previous study ([93]) and use the SIF embeddings for crime reports. However, one could train and fine-tune more recent embedding methods, like AlephBERT, specifically on police reports. This experiment will be continued in the future.

- In the scope of this thesis, we only consider homogeneous graphs where the nodes are either offenders or crimes. However, the original graph is bipartite, with nodes representing both offenders and crimes. Therefore, the bipartite graph could be converted into a hypergraph, where the nodes are offenders and a hyperedge indicates a crime and connects it to all associated offenders. In this way, both high-order semantics and complex relations between nodes can be captured, thus making transductive link prediction a hyperedge prediction problem. There is a rich literature on heterogeneous graph embedding [229–231] and even on bipartite embedding as a special case [232]. Moreover, heterogeneous multi-level frameworks have recently attempted to learn the importance of nodes through hyperedge attention mechanisms [233, 234]. An interesting future direction is to extend the DEAL framework for hyperedge prediction [235] and compare it with heterogeneous graph embedding methods and the homogeneous version.

8.3 Detecting vulnerabilities in blockchain smart contracts

To navigate through the security challenge, we started by proposing a method named MANDO, based on multi-level graph embeddings of control-flow graphs and call graphs of Solidity smart contracts to train more accurate vulnerability detection models to identify vulnerabilities in smart contracts at fine-grained line level and contract level of granularity. Our evaluation of a large-scale dataset curated from real-world Solidity smart contracts shows that our method is promising and outperforms several baselines. Our method is thus a valuable complement to other vulnerability detection techniques and contributes to smart contract security. However, with all the achievements, our MANDO framework and evaluation can still be improved further. The embedding techniques can fuse more semantic properties of the smart contract source code and adapt newer and more sophisticated graph neural networks.

Leveraging the foundation established by MANDO, MANDO-HGT is proposed as a new learning-based vulnerability detection framework for both Ethereum smart contract source code and bytecode using heterogeneous graph transformer (HGT) techniques. Inherited from MANDO, it also constructs heterogeneous contract graphs representing smart contracts' control flow and function call relations. However, instead of using metapaths as MANDO, it defines customized meta relations based on node/edge types in the generated heterogeneous graphs, then adapts HGT models to generate embeddings for nodes and graphs, and uses the embeddings to train classifiers to recognize various kinds of buggy code at the granularity levels corresponding to either individual contracts or individual lines of code. Our evaluation results in a curated smart contract dataset containing labeled vulnerabilities show that MANDO-HGT can significantly improve the accuracy of many previous vulnerability detection techniques, including best-performing learning-based and best-performing conventional analysis-based ones. The improvements in terms of the F1-score range from 0.74% to 76.89% for various bug types and detection techniques.

Additionally, we introduced a visualization tool MANDO-GURU, to demonstrate the ability of the MANDO frameworks to detect vulnerabilities in real-world blockchain smart contracts. This tool facilitates the visualization of predictive results and statistical data of analyzed smart contracts directly within web browsers. Notably, the architecture of MANDO-GURU is designed to be compatible with both MANDO and MANDO-HGT, and it is adaptable to accommodate other future MANDO frameworks with only minor adjustments.

Future works could extend the generated heterogeneous contract graphs to more comprehensive graph representations of Ethereum smart contracts (e.g., data dependencies and contract calls) for both source code and bytecode. There are several potential research directions can be explored in the future:

- Heterogeneous graph transformers could be combined with more graph learning techniques, especially those suitable for few-shot learning and

handling inconsistent labels, for more accurate and usable vulnerability detection.

- It may also be possible and interesting to utilize results from conventional testing, analysis, and verification techniques to help better train and improve our approaches.
- Although the techniques used to construct control-flow graphs and call graphs show efficiency in the MANDO, MANDO-GURU, and MANDO-HGT frameworks, they still depend on the Solidity compiler to generate the graphs. In the future, some syntax-based-only techniques, such as building abstract syntax trees first and then transforming them into control-flow graphs and call graphs, can be applied to eliminate the dependency on the Solidity compiler to extend the adaptability for forthcoming models.
- The swift advancement of large language models (LLMs), particularly for code generation, demonstrates a significant ability to model and capture the semantics of code fragments in various software programs. The extracted embedding features from LLMs could potentially represent a more semantic-aware and accurate encoding of code fragments. These embedding features could combine with advanced heterogeneous graph representation learning methods like heterogeneous graph transformers to improve the overall performance of bug detection in the future.

Appendix A

Curriculum Vitae

GENERAL INFORMATION

Full Name	Huu Hoang Nguyen
Date of Birth	21 January 1990
Place of Birth	Dong Thap, Vietnam
Address	Hannover, Germany
Phone	+49-151-400-26121
Email	ehoang@l3s.de
Homepage	https://hoanghnguyen.com
ORCID	https://orcid.org/0000-0003-0611-4634
Google Scholar	https://scholar.google.com/citations?user=cDB2Tt8AAAAJ

EDUCATION

10/2020 - 05/2024	PH.D. IN COMPUTER SCIENCE *Gottfried Wilhelm Leibniz Universität Hannover, Germany* Thesis: "Graph Representation Learning for Security Analytics in Decentralized Software Systems and Social Networks."
08/2014 - 04/2017	M.ENG. IN COMPUTER SCIENCE *Ho Chi Minh City University of Technology, Vietnam* Thesis: "Generating Control-Flow Graph from Android Binary Code."
09/2008 - 03/2013	B.SC. IN ELECTRONICS AND TELECOMMUNICATIONS *Ho Chi Minh City University of Science, Vietnam*

WORKING EXPERIENCE

02/2020 - Current	RESEARCHER *L3S Research Center, Gottfried Wilhelm Leibniz Universität Hannover, Germany* Utilizing graph embeddings to enhance investigative capabilities by predicting unseen connections in crimnal networks. Employing graph learning for vulnerability detection in blockchain smart contracts. Developing a database to manage and analyze large-scale blockchain-powered social network data.
06/2018-12/2019	RESEARCH COLLABORATOR

Ho Chi Minh City University of Technology, Vietnam
Modeling Ethereum smart contracts' control flow and data dependency. Applying machine learning to analyze Bitcoin and Ethereum transaction security vulnerabilities. Analyzing real-time data of warehouse and transportation management systems integrated with Ethereum and EOS blockchain.

05/2017-03/2018 RESEARCH ASSOCIATE
Singapore Management University, Singapore
Generating control-flow graphs and data dependencies of Android platform. Analyzing Android apps behaviors based on whole-network graphs. Context-aware code localization and recommendation.

03/2016-02/2017 RESEARCH ASSISTANT
Livelabs, Singapore Management University, Singapore
Generating control-flow graph of Android framework. Analyzing Android apps behaviors based on whole-system control flow. Identifying private data leaks in Android framework APIs.

06/2014-12/2015 ANDROID DEVELOPER
Fabrica Vietnam Co., Ltd, Vietnam
User experience analysis using Material Design. QR Code and Image Processing technologies. Payment Processing technologies. Building apps related to Coupon & Auction, Car Selling, and Overlay Photos.

01/2013-05/2014 ANDROID TEAM LEADER
EFSE Co., Ltd, Vietnam
Exploring and designing mobile apps' user-interaction interface for the young. Near Field Communication and Call Blocking technologies. Analyzing Android native launcher. Designing new techniques for floating apps. Building apps related to Android Launcher, NFC, Call Blocker, and Location.

REVIEWS

2024 Information and Software Technology journal
Elsevier, 2024

2023 Knowledge-Based Systems journal
Elsevier, 2023

2023 38th AAAI Conference on Artificial Intelligence
AAAI 2024, Vancouver, Canada, February 20-27, 2024

2023 Information and Software Technology journal
Elsevier, 2023

2023 Blockchain: Research and Applications journal
Zhejiang University Press, 2023

2023 IEEE Transactions on Multimedia journal

	IEEE, 2023
2022	37th AAAI Conference on Artificial Intelligence AAAI 2023, Washington DC, USA, February 7-24, 2023
2020	Digital Transformation and Global Society 2020 DTGS 2020, St. Petersburg, Russia, June 24-26, 2020
2017	40th International Conference on Software Engineering ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018

TECHNICAL SKILLS

Programming Languages	Python, Java, JavaScript, Solidity
Programming Frameworks	PyTorch, NetworkX, Scikit-learn, Scoot, Flask, NodeJS, KnockoutJS, D3JS
Platforms & Tools	Hive, Ethereum, Git, Android SDK, Google Cloud

ACHIEVEMENTS

2023	Two Best Paper Awards L3S Research Center, Leibniz University Hannover
2023	SIGSOFT CAPS: ICSE 2023 Travel Grants 45th International Conference on Software Engineering, ICSE 2023
2018	Silver Award $7000 Blockchain Hackathon, Vietnam Blockchain Hub 2018
2016	SMU Internship Scholarship for Excellent Graduate Students Ho Chi Minh City University of Technology
2015	500,000 app downloads Google Play Store

REFERENCES

Available upon request.

Bibliography

[1] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," *arXiv preprint arXiv:1709.05584*, 2017.

[2] W. L. Hamilton, "Graph representation learning," *Synthesis Lectures on Artifical Intelligence and Machine Learning*, vol. 14, no. 3, pp. 1–159, 2020.

[3] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*. Springer, 2006, vol. 4, no. 4.

[4] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[5] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14, 2014, pp. 701–710.

[6] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, p. 855–864.

[7] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "LINE: Large-scale information network embedding," in *Proceedings of the 24th International Conference on World Wide Web (WWW)*, 2015, pp. 1067–1077.

[8] M. Welling and T. N. Kipf, "Semi-supervised classification with graph convolutional networks," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[9] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations*, 2018.

[10] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[11] X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, and P. S. Yu, "Heterogeneous graph attention network," in *The World Wide Web Conference*, 2019, pp. 2022–2032.

[12] Z. Hu, Y. Dong, K. Wang, and Y. Sun, "Heterogeneous graph transformer," in *Proceedings of The Web Conference 2020*, 2020, pp. 2704–2710.

[13] Z. Ahmadi, H. H. Nguyen, Z. Zhang, D. Bozhkov, D. Kudenko, M. Jofre, F. Calderoni, N. Cohen, and Y. Solewicz, "Inductive and transductive link prediction for criminal network analysis," *Journal of Computational Science*, vol. 72, p. 102063, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750323001230

[14] H. H. Nguyen, D. Bozhkov, Z. Ahmadi, N.-M. Nguyen, and T.-N. Doan, "Sochaindb: A database for storing and retrieving blockchain-powered social network data," in *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '22. New York, NY, USA: Association for Computing Machinery, Jul. 2022, pp. 3036–3045. [Online]. Available: https://doi.org/10.1145/3477495.3531735

[15] T. H. Nguyen, H. H. Nguyen, Z. Ahmadi, T.-A. Hoang, and T.-N. Doan, "On the impact of dataset size: A twitter classification case study," in *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, ser. WI-IAT '21. New York, NY, USA: Association for Computing Machinery, Dec. 2021, pp. 210–217. [Online]. Available: https://doi.org/10.1145/3486622.3493960

[16] K. Maly, G. Backfried, F. Calderoni, J. Černocký, E. Dikici, M. Fabien, J. Hořínek, J. Hughes, M. Janošík, M. Kovac, P. Motlicek, H. H. Nguyen, S. Parida, J. Rohdin, M. Skácel, S. Zerr, D. Klakow, D. Zhu, and A. Krishnan, "Roxsd: a simulated dataset of communication in organized crime," in *2021 ISCA Symposium on Security and Privacy in Speech Communication*, Nov. 2021, pp. 32–36.

[17] M. Fabien, S. Parida, P. Motlícek, D. Zhu, A. Krishnan, and H. H. Nguyen, "ROXANNE research platform: Automate criminal investigations," in *Interspeech 2021, 22nd Annual Conference of the International Speech Communication Association*, H. Hermansky, H. Cernocký, L. Burget, L. Lamel, O. Scharenborg, and P. Motlícek, Eds. ISCA, Aug. 2021, pp. 962–964.

[18] H. H. Nguyen, S. Zerr, and T.-A. Hoang, "On node embedding of uncertain networks," in *2020 IEEE International Conference on Big Data (Big Data)*, Dec. 2020, pp. 5792–5794.

[19] H. H. Nguyen, N.-M. Nguyen, C. Xie, Z. Ahmadi, D. Kudendo, T.-N. Doan, and L. Jiang, "Mando-hgt: Heterogeneous graph transformers for smart contract vulnerability detection," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023, pp. 334–346.

[20] H. H. Nguyen, N.-M. Nguyen, H.-P. Doan, Z. Ahmadi, T.-N. Doan, and L. Jiang, "Mando-guru: Vulnerability detection for smart contract source code by heterogeneous graph embeddings," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the*

Foundations of Software Engineering, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 1736–1740. [Online]. Available: https://doi.org/10.1145/3540250.3558927

[21] H. H. Nguyen, N.-M. Nguyen, C. Xie, Z. Ahmadi, D. Kudendo, T.-N. Doan, and L. Jiang, "MANDO: Multi-level heterogeneous graph embeddings for fine-grained detection of smart contract vulnerabilities," in *9th IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, 2022.

[22] T. T. Nguyen, H. H. Nguyen, M. Sartipi, and M. Fisichella, "Multi-vehicle multi-camera tracking with graph based tracklet features," *IEEE Transactions on Multimedia*, pp. 1–13, 2023.

[23] ——, "Real-time multi-vehicle multi-camera tracking with graph based tracklet features," *Transportation Research Record*, May 2023. [Online]. Available: https://doi.org/10.1177/03611981231170591

[24] W. W. Zachary, "An information flow model for conflict and fission in small groups," *Journal of anthropological research*, pp. 452–473, 1977.

[25] L. C. Freeman, D. Roeder, and R. R. Mulholland, "Centrality in social networks: Ii. experimental results," *Social networks*, vol. 2, no. 2, pp. 119–141, 1979.

[26] D. Easley, J. Kleinberg *et al.*, *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge university press Cambridge, 2010, vol. 1.

[27] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[28] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 604–632, 1999.

[29] F. Calderoni, D. Brunetto, and C. Piccardi, "Communities in criminal networks: A case study," *Social Networks*, vol. 48, pp. 116–125, 2017.

[30] L. Tang and H. Liu, *Community detection and mining in social media*. Morgan & Claypool Publishers, 2010.

[31] G. Palla, I. Derényi, I. Farkas, and T. Vicsek, "Uncovering the overlapping community structure of complex networks in nature and society," *nature*, vol. 435, no. 7043, pp. 814–818, 2005.

[32] M. E. Newman, "Modularity and community structure in networks," *Proceedings of the national academy of sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.

[33] A. Ng, M. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," *Advances in neural information processing systems*, vol. 14, 2001.

[34] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the national academy of sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[35] D. Liben-Nowell and J. Kleinberg, "The link-prediction problem for social networks," *Journal of the American Society for Information Science and Technology*, vol. 58, no. 7, pp. 1019–1031, 2007.

[36] L. A. Adamic and E. Adar, "Friends and neighbors on the web," *Social Networks*, vol. 25, no. 3, pp. 211–230, 2003.

[37] A.-L. Barabâsi, H. Jeong, Z. Néda, E. Ravasz, A. Schubert, and T. Vicsek, "Evolution of the social network of scientific collaborations," *Physica A: Statistical Mechanics and Its Applications*, vol. 311, no. 3-4, pp. 590–614, 2002.

[38] L. Lü and T. Zhou, "Link prediction in complex networks: A survey," *Physica A: Statistical Mechanics and Its Applications*, vol. 390, no. 6, pp. 1150–1170, 2011.

[39] S. Soundarajan and J. Hopcroft, "Using community information to improve the precision of link prediction methods," in *Proceedings of the 21st International Conference on World Wide Web (WWW)*, 2012, pp. 607–608.

[40] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

[41] Y. Dong, N. V. Chawla, and A. Swami, "metapath2vec: Scalable representation learning for heterogeneous networks," in *the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 135–144.

[42] Minds Inc., "Minds whitepaper v2," 2021. [Online]. Available: https://cdn-assets.minds.com/front/dist/browser/en/assets/documents/Minds-Whitepaper-v2.pdf

[43] Indorse Pte. Ltd, "Indorse 2.0," 2020. [Online]. Available: https://indorse-staging-bucket.s3.amazonaws.com/Indorse+2.0+Light+Paper.pdf

[44] Allabout.me Tokens Ltd., "all.me whitepaper," 2017. [Online]. Available: https://allmestatic.com/mepaytoken/all-me_whitepaper.pdf

[45] Ethereum Foundation, "Ethereum whitepaper," 2014. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[46] Block.one, "Eos.io technical white paper v2," 2018. [Online]. Available: https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md

[47] Binance, "Binance smart chain: A parallel binance chain to enable smart contracts," 2020. [Online]. Available: https://github.com/binance-chain/whitepaper/blob/master/WHITEPAPER.md

[48] Steemit Inc., "Steem: An incentivized, blockchain-based, public content platform," 2018. [Online]. Available: https://steem.com/steem-whitepaper.pdf

[49] Hive.io, "Hive: Fast. scalable. powerful. the blockchain for web 3.0," 2020. [Online]. Available: https://hive.io/whitepaper.pdf

[50] Block.one, "Voice: The road to beta," 2019. [Online]. Available: https://b1.com/news/voice-the-road-to-beta/

[51] The BitShares Organization, "The bitshares blockchain," 2018. [Online]. Available: https://whitepaper.io/document/388/bitshares-whitepaper

[52] ARK Ecosystem, SCIC, "Ark ecosystem whitepaper," 2019. [Online]. Available: https://ark.io/Whitepaper.pdf

[53] Lisk Foundation, "Lisk consensus algorithm," 2021. [Online]. Available: https://lisk.com/documentation/lisk-sdk/protocol/consensus-algorithm.html

[54] TRON Foundation, "Tron: Advanced decentralized blockchain platform," 2018. [Online]. Available: https://tron.network/static/doc/white_paper_v_2_0.pdf

[55] N. Szabo, "The idea of smart contracts, 1 997," http://szabo.best.vwh.net/smart_contracts_idea.html, 1997, accessed: 2016-08-22. [Online]. Available: http://szabo.best.vwh.net/smart_contracts_idea.html

[56] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.

[57] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts sok," in *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. New York, NY, USA: Springer-Verlag New York, Inc., 2017, pp. 164–186. [Online]. Available: https://doi.org/10.1007/978-3-662-54455-6_8

[58] 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.

[59] 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*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 254–269. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978309

[60] D. Siegel, "Understanding the dao attack," 2016. [Online]. Available: https://www.coindesk.com/understanding-dao-hack-journalists

[61] H. Qureshi, "A hacker stole $31m of ether - how it happened, and what it means for ethereum," 2017. [Online]. Available: https://medium.freecodecamp.org/

[62] J. Baumgartner, S. Zannettou, B. Keegan, M. Squire, and J. Blackburn, "The pushshift reddit dataset," in *Proceedings of the International AAAI Conference on Web and Social Media*, vol. 14, 2020, pp. 830–839.

[63] J. Baumgartner, S. Zannettou, M. Squire, and J. Blackburn, "The pushshift telegram dataset," in *Proceedings of the International AAAI Conference on Web and Social Media*, vol. 14, 2020, pp. 840–847.

[64] J. J. McAuley and J. Leskovec, "Learning to discover social circles in ego networks." in *NIPS*, vol. 2012. Citeseer, 2012, pp. 548–56.

[65] E. Cho, S. A. Myers, and J. Leskovec, "Friendship and mobility: user movement in location-based social networks," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2011, pp. 1082–1090.

[66] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[67] A. S. M. Tayeen, A. Mtibaa, and S. Misra, "Location, location, location! quantifying the true impact of location on business reviews using a yelp dataset," in *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, 2019, pp. 1081–1088.

[68] C. S. Bojer and J. P. Meldgaard, "Kaggle forecasting competitions: An overlooked learning opportunity," *International Journal of Forecasting*, vol. 37, no. 2, pp. 587–603, 2021.

[69] M. Mohri and A. M. Medina, "Learning theory and algorithms for revenue optimization in second price auctions with reserve," in *International Conference on Machine Learning*. PMLR, 2014, pp. 262–270.

[70] A. Capocci, V. D. Servedio, F. Colaiori, L. S. Buriol, D. Donato, S. Leonardi, and G. Caldarelli, "Preferential attachment in the growth of social networks: The internet encyclopedia wikipedia," *Physical review E*, vol. 74, no. 3, p. 036116, 2006.

[71] C. B. Clement, M. Bierbaum, K. P. O'Keeffe, and A. A. Alemi, "On the use of arxiv as a dataset," *arXiv preprint arXiv:1905.00075*, 2019.

[72] K. Oliver, N. Crossley, G. Edwards, J. Koskinen, M. Everett, and C. Broccatelli, "Covert networks: structures, processes and types," *Unpublished manuscript, University of Manchester, Manchester, UK*, pp. 4–13, 2014.

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

[74] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: a framework to analyze solidity smart contracts," in *the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1349–1352.

[75] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[76] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network." in *IJCAI*, 2020, pp. 3283–3290.

[77] 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.

[78] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "ethor: Practical and provably sound static analysis of ethereum smart contracts," in *the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 621–640.

[79] C. Li and B. Palanisamy, "Incentivized blockchain-based social media platforms: A case study of steemit," in *Proceedings of the 10th ACM Conference on Web Science*, 2019, pp. 145–154.

[80] C. Li, B. Palanisamy, R. Xu, J. Xu, and J. Wang, "Steemops: Extracting and analyzing key operations in steemit blockchain-based social media platform," *arXiv preprint arXiv:2102.00177*, 2021.

[81] G. M. Campedelli, *Machine Learning for Criminology and Crime Research: At the Crossroads*. London: Routledge, 2022.

[82] J. Woodhams and K. Toye, "An empirical test of the assumptions of case linkage and offender profiling with serial commercial robberies." *Psychology, Public Policy, and Law*, vol. 13, no. 1, p. 59, 2007.

[83] C. Bennell, B. Snook, S. Macdonald, J. C. House, and P. J. Taylor, "Computerized crime linkage systems: A critical review and research agenda," *Criminal Justice and Behavior*, vol. 39, no. 5, pp. 620–634, 2012.

[84] Y. Li and X. Shao, "Thresholds learning of three-way decisions in pairwise crime linkage," *Applied Soft Computing*, vol. 120, p. 108638, 2022.

[85] Y.-S. Li, H. Chi, X.-Y. Shao, M.-L. Qi, and B.-G. Xu, "A novel random forest approach for imbalance problem in crime linkage," *Knowledge-Based Systems*, vol. 195, p. 105738, 2020.

[86] H. Chi, Z. Lin, H. Jin, B. Xu, and M. Qi, "A decision support system for detecting serial crimes," *Knowledge-Based Systems*, vol. 123, pp. 88–101, 2017.

[87] J. Vimala Devi and K. Kavitha, "Adaptive deep q learning network with reinforcement learning for crime prediction," *Evolutionary Intelligence*, pp. 1–12, 2022.

[88] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[89] B. Wang, D. Zhang, D. Zhang, P. Brantingham, and A. L. Bertozzi, "Deep learning for real time crime forecasting," *IEICE Proceeding Series*, vol. 29, no. A3L-E-2-4, pp. 330–333, 2017.

[90] Q. Wang, G. Jin, X. Zhao, Y. Feng, and J. Huang, "Csan: A neural network benchmark model for crime forecasting in spatio-temporal scale," *Knowledge-Based Systems*, vol. 189, p. 105120, 2020.

[91] K. Simonyan and A. Zisserman, "Two-stream convolutional networks for action recognition in videos," *Advances in Neural Information Processing Systems*, vol. 27, 2014.

[92] Z. Wu, X. Wang, Y.-G. Jiang, H. Ye, and X. Xue, "Modeling spatial-temporal clues in a hybrid deep learning framework for video classification," in *Proceedings of the 23rd ACM International Conference on Multimedia*, 2015, pp. 461–470.

[93] A. Solomon, A. Magen, S. Hanouna, M. Kertis, B. Shapira, and L. Rokach, "Crime linkage based on textual hebrew police reports utilizing behavioral patterns," in *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM)*, 2020, p. 2749–2756.

[94] A. Ghazvini, S. N. H. S. Abdullah, M. K. Hasan, and D. Z. A. B. Kasim, "Crime spatiotemporal prediction with fused objective function in time delay neural network," *IEEE Access*, vol. 8, pp. 115 167–115 183, 2020.

[95] A. Bojchevski and S. Günnemann, "Deep gaussian embedding of graphs: Unsupervised inductive learning via ranking," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

[96] S. Jha, E. Yang, A. O. Almagrabi, A. K. Bashir, and G. P. Joshi, "Comparative analysis of time series model and machine testing systems for crime forecasting," *Neural Computing and Applications*, vol. 33, no. 17, pp. 10 621–10 636, 2021.

[97] T. Nakaya and K. Yano, "Visualising crime clusters in a space-time cube: An exploratory data-analysis approach using space-time kernel density estimation and scan statistics," *Transactions in GIS*, vol. 14, no. 3, pp. 223–239, 2010.

[98] Y.-L. Lin, M.-F. Yen, and L.-C. Yu, "Grid-based crime prediction using geographical features," *ISPRS International Journal of Geo-Information*, vol. 7, no. 8, p. 298, 2018.

[99] X. Zhang, L. Liu, L. Xiao, and J. Ji, "Comparison of machine learning algorithms for predicting crime hotspots," *IEEE Access*, vol. 8, pp. 181 302–181 310, 2020.

[100] A. M. Olligschlaeger, "Artificial neural networks and crime mapping," *Crime Mapping and Crime Prevention*, vol. 1, p. 313, 1997.

[101] W. Gorr, A. Olligschlaeger, and Y. Thompson, "Short-term forecasting of crime," *International Journal of Forecasting*, vol. 19, no. 4, pp. 579–594, 2003.

[102] M. Feng, J. Zheng, J. Ren, A. Hussain, X. Li, Y. Xi, and Q. Liu, "Big data analytics and mining for effective visualization and trends forecasting of crime data," *IEEE Access*, vol. 7, pp. 106 111–106 123, 2019.

[103] K. Kianmehr and R. Alhajj, "Effectiveness of support vector machine for crime hot-spots prediction," *Applied Artificial Intelligence*, vol. 22, no. 5, pp. 433–458, 2008.

[104] C.-H. Yu, M. W. Ward, M. Morabito, and W. Ding, "Crime forecasting using data mining techniques," in *Proceedings of the IEEE 11th International Conference on Data Mining Workshops (ICDMW)*, 2011, pp. 779–786.

[105] E. Eftelioglu, S. Shekhar, J. M. Kang, and C. C. Farah, "Ring-shaped hotspot detection," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 12, pp. 3367–3381, 2016.

[106] G. O. Mohler, M. B. Short, P. J. Brantingham, F. P. Schoenberg, and G. E. Tita, "Self-exciting point process modeling of crime," *Journal of the American Statistical Association*, vol. 106, no. 493, pp. 100–108, 2011.

[107] J. H. Ratcliffe, "A temporal constraint theory to explain opportunity-based spatial offending patterns," *Journal of Research in Crime and Delinquency*, vol. 43, no. 3, pp. 261–291, 2006.

[108] M. Rosenblatt, "Remarks on some nonparametric estimates of a density function," *The Annals of Mathematical Statistics*, pp. 832–837, 1956.

[109] B. W. Silverman, *Density estimation for statistics and data analysis*. Routledge, 2018.

[110] J. L. Toole, N. Eagle, and J. B. Plotkin, "Spatiotemporal correlations in criminal offense records," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 4, pp. 1–18, 2011.

[111] M. Alharby, A. Aldweesh, and A. v. Moorsel, "Blockchain-based smart contracts: A systematic mapping study of academic research," in *International Conference on Cloud Computing, Big Data and Blockchain*, 2018, pp. 1–6.

[112] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *the 33rd IEEE/ACM International Conference on Automated Software Engineering*, 2018, pp. 259–269.

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

[114] I. Ashraf, X. Ma, B. Jiang, and W. K. Chan, "GasFuzzer: Fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities," *IEEE Access*, vol. 8, pp. 99 552–99 564, 2020.

[115] Y. Liu, Y. Li, S.-W. Lin, and Q. Yan, "ModCon: A model-based testing platform for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1601–1605.

[116] Y. Huang, B. Jiang, and W. K. Chan, "EOSFuzzer: Fuzzing eosio smart contracts for vulnerability detection," in *12th Asia-Pacific Symposium on Internetware*, 2020, pp. 99–109.

[117] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *the 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 1186–1189.

[118] B. Jiang, Y. Chen, D. Wang, I. Ashraf, and W. Chan, "WANA: Symbolic execution of wasm bytecode for extensible smart contract vulnerability detection," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2021, pp. 926–937.

[119] K. Weiss and J. Schütte, "Annotary: A concolic execution system for developing secure smart contracts," in *European Symposium on Research in Computer Security*. Springer, 2019, pp. 747–766.

[120] S. So, S. Hong, and H. Oh, "*smartest*: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution," in *30th USENIX Security Symposium*, 2021, pp. 1361–1378.

[121] Consensys, "Mythril framework," 2017. [Online]. Available: https://github.com/ConsenSys/mythril

[122] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng, "Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1029–1040.

[123] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2019, pp. 8–15.

[124] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and tools for the static analysis of ethereum smart contracts," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 51–78.

[125] ——, "Ethertrust: Sound static analysis of ethereum bytecode," *Technische Universität Wien, Tech. Rep*, 2018.

[126] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *25th ACM Conference on Computer and Communications Security*, 2018.

[127] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *the ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[128] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of ethereum smart contracts," in *the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[129] A. Wang, H. Wang, B. Jiang, and W. K. Chan, "Artemis: An improved smart contract verification tool for vulnerability detection," in *2020 7th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 2020, pp. 173–181.

[130] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," *ACM Computing Surveys (CSUR)*, vol. 54, no. 7, pp. 1–38, 2021.

[131] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu, "A formal verification tool for ethereum vm bytecode," in *26th ACM ESEC/FSE*, 2018, pp. 912–915.

[132] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE symposium on security and privacy (SP)*. IEEE, 2020, pp. 1661–1677.

[133] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu *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.

[134] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun, "Semantic understanding of smart contracts: Executable operational semantics of solidity," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1695–1712.

[135] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *International Conference on Principles of Security and Trust*. Springer, 2018, pp. 243–269.

[136] I. Garfatta, K. Klai, W. Gaaloul, and M. Graiet, "A survey on formal verification for solidity smart contracts," in *2021 Australasian Computer Science Week Multiconference*, 2021, pp. 1–10.

[137] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *The Network and Distributed System Security Symposium*, 2018.

[138] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "VulDeeLocator: a deep learning-based fine-grained vulnerability detector," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[139] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection," *Information and Software Technology*, vol. 136, p. 106576, 2021.

[140] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *17th IEEE international conference on machine learning and applications (ICMLA)*, 2018, pp. 757–762.

[141] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui, "Static detection of control-flow-related vulnerabilities using graph embedding," in *24th ICECCS*. IEEE, 2019, pp. 41–50.

[142] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[143] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.

[144] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "Vulcnn: An image-inspired scalable vulnerability detection system," in *ICSE*, 2022.

[145] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[146] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "DeepWukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM TOSEM*, vol. 30, no. 3, pp. 1–33, 2021.

[147] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 292–303.

[148] K. Zhang, W. Wang, H. Zhang, G. Li, and Z. Jin, "Learning to represent programs with heterogeneous graphs," in *ICPC*, 2022.

[149] S. Han, B. Liang, J. Huang, and W. Shi, "DC-Hunter: Detecting dangerous smart contracts via bytecode matching," *Journal of Cyber Security*, May 2020.

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

[151] Z. Bo, S. Chenhan, P. Xiaoyan, A. Yang, T. Juncheng, and Y. Anqi, "Semantic-aware graph neural network for smart contract bytecode vulnerability detection," *Advanced Engineering Sciences*, vol. 54, no. 2, pp. 49–55, 2022.

[152] D. Zhu, F. Yue, J. Pang, X. Zhou, W. Han, and F. Liu, "Bytecode similarity detection of smart contract across optimization options and compiler versions based on triplet network," *Electronics*, vol. 11, no. 4, p. 597, 2022. [Online]. Available: https://github.com/Zdddzz/smartcontract

[153] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "ContractWard: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1133–1144, 2020.

[154] A. K. Gogineni, S. Swayamjyoti, D. Sahoo, K. K. Sahu, and R. Kishore, "Multi-class classification of vulnerabilities in smart contracts using AWD-LSTM, with pre-trained encoder inspired from natural language processing," *IOP SciNotes*, vol. 1, no. 3, p. 035002, 2020.

[155] T. H.-D. Huang, "Hunting the ethereum smart contract: Color-inspired inspection of potential attacks," *arXiv preprint arXiv:1807.01868*, 2018.

[156] O. Lutz, H. Chen, H. Fereidooni, C. Sendner, A. Dmitrienko, A. R. Sadeghi, and F. Koushanfar, "ESCORT: Ethereum smart contracts vulnerability detection using deep neural network and transfer learning," *arXiv preprint arXiv:2103.12607*, 2021.

[157] W. J.-W. Tann, X. J. Han, S. S. Gupta, and Y.-S. Ong, "Towards safer smart contracts: A sequence learning approach to detecting security threats," *arXiv preprint arXiv:1811.06632*, 2018.

[158] J. Tian, W. Xing, and Z. Li, "Bvdetector: A program slice-based binary code vulnerability intelligent detection system," *Information and Software Technology*, vol. 123, p. 106289, 2020.

[159] T. Le, T. Nguyen, T. Le, D. Phung, P. Montague, O. De Vel, and L. Qu, "Maximal divergence sequential autoencoder for binary software vulnerability detection," in *International Conference on Learning Representations (ICLR)*, 2018.

[160] Y. J. Lee, S.-H. Choi, C. Kim, S.-H. Lim, and K.-W. Park, "Learning binary code with deep learning to detect software weakness," in *KSII the 9th international conference on internet (ICONI) 2017 symposium*, 2017.

[161] H. Hanif, M. H. N. M. Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar, "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches," *Journal of Network and Computer Applications*, vol. 179, p. 103009, 2021.

[162] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: semantic-aware neural networks for binary code similarity detection," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, 2020, pp. 1145–1152.

[163] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He, and S. Ji, "Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion," *arXiv preprint arXiv:2106.09282*, 2021.

[164] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang, and X. Mao, "Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques," in *the 32nd International Symposium on Software Reliability Engineering*, 2021.

[165] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Transactions on Software Engineering*, 2020.

[166] S. Jeon, G. Lee, H. Kim, and S. S. Woo, "Smartcondetect: Highly accurate smart contract code vulnerability detection mechanism using bert," in *KDD Workshop on Programming Language Processing*, 2021.

[167] H. Zhao, P. Su, Y. Wei, K. Gai, and M. Qiu, "Gan-enabled code embedding for reentrant vulnerabilities detection," in *Knowledge Science, Engineering and Management*, 2021, pp. 585–597.

[168] Facebook, "Facebook press release," 2021. [Online]. Available: https://investor.fb.com/investor-news/

[169] Twitter, "Twitter about," 2021. [Online]. Available: https://about.twitter.com

[170] Coin Market Cap, "Hive price," 2021. [Online]. Available: https://coinmarketcap.com/currencies/hive-blockchain/

[171] P. Baker, "Steem hard fork confiscates $6.3m, community immediately takes it back," 2020. [Online]. Available: https://www.coindesk.com/steem-hard-fork-hive

[172] J. Redman, "Bitcoin fees tap $60 per transaction, users say fees restrict adoption, others 'embrace' the btc fee pump," 2021. [Online]. Available: https://news.bitcoin.com/

[173] PeakD, "Peakd," 2021. [Online]. Available: https://peakd.com

[174] HiveBlog, "Hiveblog," 2021. [Online]. Available: https://hive.blog

[175] J. Tigani and S. Naidu, *Google BigQuery Analytics*. John Wiley & Sons, 2014.

[176] Falcon Framework, "Falcon framework," 2021. [Online]. Available: https://falconframework.org/

[177] F. Gimian, "Choosing a fast python api framework," 2018. [Online]. Available: https://fgimian.github.io/blog/2018/05/17/choosing-a-fast-python-api-framework/

[178] G. Csányi and B. Szendrői, "Structure of a large social network," *Physical Review E*, vol. 69, no. 3, p. 036131, 2004.

[179] D. Weisburd, "The law of crime concentration and the criminology of place," *Criminology*, vol. 53, no. 2, pp. 133–157, 2015.

[180] N. N. Martinez, Y. Lee, J. E. Eck, and S. O, "Ravenous wolves revisited: a systematic review of offending concentration," *Crime Science*, vol. 6, no. 1, p. 10, 2017.

[181] Y. Lee, J. E. Eck, S. O, and N. N. Martinez, "How concentrated is crime at places? A systematic review from 1970 to 2015," *Crime Science*, vol. 6, no. 1, p. 6, 2017.

[182] M. A. Tayebi, U. Glässer, M. A. Tayebi, and U. Glässer, *Social network analysis in predictive policing*. Springer, 2016.

[183] A. G. Ferguson, *Policing Predictive Policing*. Washington University Law Review, 2017, vol. 94, no. 5.

[184] K. J. Bowers and S. D. Johnson, "Who commits near repeats? a test of the boost explanation," *Western Criminology Review*, vol. 5, no. 3, 2004.

[185] S. D. Johnson, W. Bernasco, K. J. Bowers, H. Elffers, J. Ratcliffe, G. Rengert, and M. Townsley, "Space–time patterns of risk: A cross national assessment of residential burglary victimization," *Journal of Quantitative Criminology*, vol. 23, no. 3, pp. 201–219, 2007.

[186] O. Kounadi, A. Ristea, A. Araujo, and M. Leitner, "A systematic review on spatial crime forecasting," *Crime Science*, vol. 9, no. 1, p. 7, 2020.

[187] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[188] H. Zhang, "The optimality of naive bayes," in *Proceedings of the 17th International FLAIRS Conference (FLAIRS)*, 2004, pp. 562–567.

[189] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[190] C. D. Uchida, "Predictive Policing," in *Encyclopedia of Criminology and Criminal Justice*, G. Bruinsma and D. Weisburd, Eds. Springer, 2014, pp. 3871–3880.

[191] A. Meijer and M. Wessels, "Predictive Policing: Review of Benefits and Drawbacks," *International Journal of Public Administration*, vol. 42, no. 12, pp. 1031–1039, 2019.

[192] J. H. Ratcliffe, "Advocate: Predictive Policing," in *Police Innovation: Contrasting Perspectives*, 2nd ed., D. Weisburd and A. A. Braga, Eds. Cambridge: Cambridge University Press, 2019, pp. 347–365.

[193] R. Boba Santos, "Critic: Predictive Policing: Where's the Evidence?" in *Police Innovation: Contrasting Perspectives*, 2nd ed., D. Weisburd and A. A. Braga, Eds. Cambridge University Press, 2019, pp. 366–398.

[194] Y.-S. Li and M.-L. Qi, "An approach for understanding offender modus operandi to detect serial robbery crimes," *Journal of Computational Science*, vol. 36, p. 101024, 2019.

[195] K. Davies and J. Woodhams, "The practice of crime linkage: A review of the literature," *Journal of Investigative Psychology and Offender Profiling*, vol. 16, no. 3, pp. 169–200, 2019.

[196] J. Woodhams, C. R. Hollin, and R. Bull, "The psychology of linking crimes: A review of the evidence," *Legal and Criminological Psychology*, vol. 12, no. 2, pp. 233–249, 2007.

[197] P. Stalidis, T. Semertzidis, and P. Daras, "Examining deep learning architectures for crime classification and prediction," *Forecasting*, vol. 3, no. 4, pp. 741–762, 2021.

[198] R. N. Lichtenwalter, J. T. Lussier, and N. V. Chawla, "New perspectives and methods in link prediction," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2010, pp. 243–252.

[199] M. A. Hasan, V. Chaoji, S. Salem, and M. Zaki, "Link prediction using supervised learning," in *Proceedings of the SIAM Data Mining Workshop on Link Analysis, Counterterrorism and Security*, 2006.

[200] M. A. Tayebi, M. Ester, U. Glässer, and P. L. Brantingham, "Spatially embedded co-offence prediction using supervised learning," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2014, p. 1789–1798.

[201] S. Arora, Y. Liang, and T. Ma, "A simple but tough-to-beat baseline for sentence embeddings," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[202] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 2019, pp. 4171–4186.

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

[204] Y. Hao, X. Cao, Y. Fang, X. Xie, and S. Wang, "Inductive link prediction for nodes having only attribute information," in *Proceedings of the 29th International Joint Conferences on Artificial Intelligence (IJCAI)*, 2021, pp. 1209–1215.

[205] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 2, 2006, pp. 1735–1742.

[206] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and Édouard Duchesnay, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. 85, pp. 2825–2830, 2011. [Online]. Available: http://jmlr.org/papers/v12/pedregosa11a.html

[207] C. K. Williams and C. E. Rasmussen, *Gaussian processes for machine learning*. MIT press Cambridge, MA, 2006, vol. 2, no. 3.

[208] A. Zhou, J. Yang, Y. Gao, T. Qiao, Y. Qi, X. Wang, Y. Chen, P. Dai, W. Zhao, and C. Hu, "Brief industry paper: Optimizing memory efficiency of graph neural networks on edge computing platforms," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 445–448.

[209] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[210] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–43, 2020.

[211] Y. Sun, J. Han, X. Yan, P. S. Yu, and T. Wu, "Pathsim: Meta path-based top-k similarity search in heterogeneous information networks," *the VLDB Endowment*, vol. 4, no. 11, pp. 992–1003, 2011.

[212] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[213] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[214] B. Mueller, "Smashing smart contracts for fun and real profit," in *9th annual HITB Security Conference*, 2018, pp. 2–51.

[215] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[216] J. Frizzo-Barker, P. A. Chow-White, P. R. Adams, J. Mentanko, D. Ha, and S. Green, "Blockchain as a disruptive technology for business: A systematic review," *International Journal of Information Management*, vol. 51, p. 102029, 2020.

[217] B. Bhushan, P. Sinha, K. M. Sagayam, and J. Andrew, "Untangling blockchain technology: A survey on state of the art, security threats, privacy services, applications and future research directions," *Computers & Electrical Engineering*, vol. 90, p. 106897, 2021.

[218] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintenance-related concerns for post-deployed ethereum smart contract development: issues, techniques, and future challenges," *Empirical Software Engineering*, vol. 26, no. 6, pp. 1–44, 2021.

[219] Y. Sun and J. Han, "Mining heterogeneous information networks: a structural analysis approach," *Acm Sigkdd Explorations Newsletter*, vol. 14, no. 2, pp. 20–28, 2013.

[220] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," *Advances in neural information processing systems*, vol. 30, 2017.

[221] A. Duval and F. Malliaros, "Graphsvx: Shapley value explanations for graph neural networks," in *European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, 2021.

[222] F. Contro, M. Crosara, M. Ceccato, and M. Dalla Preda, "Ethersolve: Computing an accurate control-flow graph from ethereum bytecode," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 127–137.

[223] Crytic-compile, "Abstraction layer for smart contract build systems," https://github.com/crytic/crytic-compile, 2022.

[224] F. Contro, M. Crosara, and cmariano, "SeUniVr/EtherSolve: Version used for ICPC-2021 paper," Mar. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4607305

[225] L. S. Shapley, *17. A Value for n-Person Games*. Princeton: Princeton University Press, 2016, pp. 307–318. [Online]. Available: https://doi.org/10.1515/9781400881970-018

[226] S. Ramírez, "Fastapi framework, high performance, easy to learn, fast to code, ready for production," Berlin, Germany, 2022. [Online]. Available: https://fastapi.tiangolo.com/

[227] Meta Platforms, Inc., "React: A javascript library for building user interfaces," 2022. [Online]. Available: https://reactjs.org/

[228] ApexCharts, "Apexcharts.js: Modern & interactive open-source charts," 2022. [Online]. Available: https://apexcharts.com/

[229] Y. Dong, N. V. Chawla, and A. Swami, "metapath2vec: Scalable representation learning for heterogeneous networks," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 135–144.

[230] T.-y. Fu, W.-C. Lee, and Z. Lei, "Hin2vec: Explore meta-paths in heterogeneous information networks for representation learning," in *Proceedings of the ACM on Conference on Information and Knowledge Management (CIKM)*, 2017, pp. 1797–1806.

[231] R. Aponte, R. A. Rossi, S. Guo, J. Hoffswell, N. Lipka, C. Xiao, G. Chan, E. Koh, and N. Ahmed, "A hypergraph neural network framework for learning hyperedge-dependent node embeddings," *arXiv preprint arXiv:2212.14077*, 2022.

[232] W. Huang, Y. Li, Y. Fang, J. Fan, and H. Yang, "Biane: Bipartite attributed network embedding," in *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2020, pp. 149–158.

[233] H. Fan, F. Zhang, Y. Wei, Z. Li, C. Zou, Y. Gao, and Q. Dai, "Heterogeneous hypergraph variational autoencoder for link prediction," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 8, pp. 4125–4138, 2021.

[234] H. Chen, H. Yin, X. Sun, T. Chen, B. Gabrys, and K. Musial, "Multi-level graph convolutional networks for cross-platform anchor link prediction," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 1503–1511.

[235] C. Chen and Y.-Y. Liu, "A survey on hyperlink prediction," *arXiv preprint arXiv:2207.02911*, 2022.