AI-Augmented Vulnerability Detection and Patching

by

Poornaditya Mishra

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
(Artificial Intelligence)
in The University of Michigan-Dearborn
2024

Thesis Committee:

Associate Professor Birhanu Eshete, Chair
Assistant Professor Zheng Song
Professor Bruce Maxim

Dedication

To my mother, my unwavering pillar of strength and the guiding light of my life. Your endless love, unwavering support, and unwavering belief in me have carried me through every challenge and lifted me to new heights. You have been there for me at my lowest, celebrating my triumphs, and providing solace during setbacks. It is through your guidance and inspiration that I have become the person I am today. Thank you for your sacrifices and for all that you have done for me. This thesis, and every achievement it represents, is dedicated to you.

ACKNOWLEDGEMENTS

This thesis would not have been possible without the support and guidance of many individuals. First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Birhanu Eshete, for his unwavering support, encouragement, and mentorship throughout my thesis journey. His invaluable guidance, insightful feedback, and ability to push me beyond my perceived limits were instrumental in shaping this work and allowing me to grow as a researcher. I am sincerely grateful to Dr. Jin Lu for igniting the spark of this research. His initial guidance and support were invaluable in helping me take the first steps on this path. My sincere thanks go to Dr. Probir Roy, who opened my eyes to the fascinating world of Abstract Syntax Trees and Intermediate Representations. His insights and enthusiasm in this area were truly inspiring. I would also like to thank my friends and family for their constant love and support, providing a much-needed source of strength and motivation throughout the challenging times. To everyone who contributed to this thesis in ways big and small, thank you from the bottom of my heart.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

AI-Augmented Vulnerability Detection and Patching

by

Poornaditya Mishra

Chair: Dr. Birhanu Eshete

Software vulnerabilities remain a persistent threat, and the increasing use of AI-generated code introduces new security challenges. While Large Language Models (LLMs) excel at code generation, they often struggle to consistently produce secure code or apply targeted vulnerability fixes. This work proposes a novel system that bridges this gap by combining the strengths of graph-based deep learning and LLMs for automated vulnerability detection and patching. We first model vulnerability detection as graph representation learning via Graph Attention Network (GAT) to accurately identify vulnerabilities in code, leveraging the rich structural information encoded in Code Property Graphs (CPGs) and Abstract Syntax Trees (ASTs). Our system then leverages the GAT's predictions to guide an LLM, providing both the vulnerability type and the precise location within the code requiring a patch. This targeted guidance enables the LLM to generate more secure and contextually appropriate code modifications. Through experiments on a dataset of real-world vulnerable code, we demonstrate the effectiveness of our approach in detecting critical vulnerabilities like SQL injection and session hijacking. We further evaluate the quality of the LLM-generated patches, showing a significant improvement in security when guided by our system. This research paves the way for more secure and reliable AI-assisted software development by integrating deep learning-based vulnerability analysis with the generative capabilities of LLMs.

Keywords: Generative AI, Cybersecurity, Large Language Model

CHAPTER I

Introduction

1.1 Motivation

The pervasiveness of software in modern society is undeniable. It underpins vital services such as healthcare and finance, as well as everyday conveniences like communication and entertainment. As our reliance on software continues to grow exponentially, so do the potential risks associated with security vulnerabilities. Exploitable weaknesses within code can have devastating consequences, leading to data breaches, financial losses, system failures, and disruption of critical infrastructure and services.

While AI-powered code generation tools hold immense promise for increasing productivity and efficiency in software development, they also introduce new challenges to ensuring code security. Large Language Models (LLMs), with their remarkable ability to generate human-quality code, often lack the ability to consistently produce secure code. Trained on massive datasets of code, LLMs may inadvertently learn and reproduce insecure coding patterns, potentially introducing vulnerabilities or failing to recognize and address existing ones. This highlights a critical need for techniques that can guide LLMs towards generating more secure code and assist developers in identifying and patching vulnerabilities, ensuring the development of robust and reliable software systems.

1.2 Prior Work

Traditional approaches to vulnerability detection, while essential in software development, often struggle to keep pace with the increasing complexity and scale of modern software. Manual code review, considered the gold standard for identifying vulnerabilities [1], involves meticulous inspection of source code by security experts. While valuable, this process is inherently time-consuming, resource-intensive, and susceptible to human error, especially as codebases grow in size and complexity.

Static analysis tools offer a degree of automation by analyzing source code without execution, searching for patterns and anomalies that might indicate vulnerabilities [2, 3]. While these tools offer efficiency and broad coverage, they are often plagued by false positives, flagging code that is not actually vulnerable, leading to wasted effort in manual verification. Furthermore, their reliance on predefined patterns and rules limits their contextual awareness, often failing to capture vulnerabilities that arise from the subtle interplay of multiple code components or those requiring a deeper understanding of program execution behavior and semantics.

Fuzzing takes a dynamic approach, providing invalid or unexpected inputs to a program to trigger crashes or unusual behavior that might reveal vulnerabilities [4, 5]. While effective at uncovering certain types of vulnerabilities, especially those related to memory corruption, fuzzing can be computationally expensive and struggles to detect logic-based vulnerabilities, such as authentication bypasses or flaws in business logic.

The rise of deep learning has introduced new possibilities for vulnerability detection [6]. However, many existing deep learning approaches[7, 8, 9] treat code as plain text, akin to natural language processing, failing to fully capture the intricate structural and semantic relationships that are crucial for understanding code behavior and identifying security vulnerabilities. This limitation hinders their effectiveness in identifying vulnerabilities that stem from the complex interplay of code elements and their dependencies.

Graph Neural Networks (GNNs) have emerged as a promising solution to address this challenge. Designed to learn from data represented as graphs, GNNs are particularly well-suited for code analysis as they can effectively capture the complex dependencies within code [10, 11]. Specifically, Graph Attention Networks (GATs) [12], a type of GNN with an attention mechanism, have shown remarkable capabilities in learning from graph-structured data, selectively focusing on the most relevant connections within the code. Research has demonstrated the effectiveness of GNNs in learning from code representations like Code Property Graphs (CPGs) [13], which encode not only syntactic information but also data flow, control flow, and other important semantic relationships.

Large Language Models (LLMs), with their impressive code generation capabilities, offer a potential avenue for automated code repair [14]. However, simply prompting an LLM to "fix vulnerabilities" without informed guidance often yields sub-optimal results. LLMs, while adept at generating syntactically correct code, often lack the ability to understand the security implications of the code they generate. Therefore, they require careful context and explicit instructions regarding security considerations to generate patches that are both functionally correct and secure.

1.3 Approach Overview

This thesis proposes a novel framework that seamlessly integrates the power of GATs for precise vulnerability detection with the generative capabilities of LLMs for automated vulnerability patching. Recognizing the limitations of relying solely on LLMs for security-critical tasks, our approach leverages GATs to not only identify vulnerabilities but also to provide targeted guidance to an LLM, thereby ensuring the generation of secure and effective patches.

The core insight of our approach lies in representing code as Abstract Syntax

Trees (ASTs) and Code Property Graphs (CPGs), capturing the hierarchical structure and syntactic relationships in code. We train GAT models on AST and CPG representations, enabling them to learn and recognize complex patterns indicative of vulnerabilities. The GAT's attention mechanism plays a crucial role in precisely pinpointing vulnerable code sections by highlighting the nodes most relevant to its vulnerability prediction.

This localized vulnerability information, enriched with relevant contextual data extracted from the AST, forms the basis for crafting carefully engineered prompts for a powerful LLM (Google Gemini Pro). These prompts, unlike generic instructions, provide the LLM with specific guidance on the vulnerability's nature and location within the code, guiding it towards generating effective patches that address the identified issue while preserving the original code's functionality.

While LLMs exhibit remarkable code generation capabilities, relying solely on them for security-critical tasks presents significant challenges. Their training on vast codebases, while advantageous for code fluency, inadvertently exposes them to both secure and insecure coding practices. This can lead to the unintentional generation of code susceptible to common vulnerabilities, even from seemingly innocuous prompts.

For instance, an LLM tasked with creating a simple web application might produce functional code that lacks essential security measures like input sanitization or secure session management. This vulnerability could stem from the model learning prevalent, but insecure, coding patterns from its training data. While the generated code might function as intended on the surface, it would remain vulnerable to common exploits like cross-site scripting (XSS) or SQL injection.

Furthermore, simply instructing an LLM to "improve security" without specific guidance often results in superficial fixes or overly broad security measures. This lack of targeted guidance stems from the LLM's inability to independently analyze the code for specific vulnerabilities and devise appropriate remediation strategies.

This highlights a critical need to guide LLMs toward secure code generation through a more structured and informative approach. Our proposed framework addresses this challenge by incorporating:

1. **Precise Vulnerability Detection:** Leveraging the power of GATs to analyze code structure and identify specific vulnerabilities with high accuracy.

2. **Targeted Vulnerability Localization:** the attention mechanism of GATs to pinpoint the exact code segments requiring remediation.

3. **Context-Aware Prompt Engineering:** Generating LLM prompts enriched with localized vulnerability information and relevant code context to guide security patch generation.

1.4 Evalaution Overview

We rigorously evaluate our framework through experiments on a dataset of 16,000 real-world Python code snippets, focusing on common web application vulnerabilities such as SQL Injection, Cross-Site Scripting, Command Injection, Path Traversal, and Insecure Session Management. Our evaluation encompasses a multi-faceted assessment to gauge the effectiveness of our proposed approach:

- **Vulnerability Detection Accuracy:** We assess the accuracy of our GAT model in detecting vulnerabilities using standard classification metrics, including accuracy, precision, recall, and F1-score. Our experiments reveal that leveraging Code Property Graphs (CPGs) as the code representation significantly enhances vulnerability detection accuracy compared to using Abstract Syntax Trees (ASTs).

- **Localization Effectiveness:** We evaluate the effectiveness of our attention-based localization technique in precisely pinpointing vulnerable code sections.

Our analysis demonstrates an 87% accuracy in identifying the vulnerable code span, highlighting the efficacy of this approach in guiding the subsequent patch generation process.

- **Patch Quality Assessment:** We scrutinize the quality of the LLM-generated patches using a combination of human evaluation, static analysis with Bandit [15], and re-evaluation with our trained GAT model. Our evaluation reveals a promising success rate of around 75-80% in generating correct and effective patches, demonstrating the potential of LLMs for automated code repair when guided by our framework.

The findings from our evaluation underscore the significance of this thesis in demonstrating the viability of a hybrid approach, combining graph-based deep learning and LLMs, for automated vulnerability detection and patching. Our work highlights the importance of choosing a suitable code representation, the effectiveness of attention-based localization for targeted patch generation, and the promising capabilities of LLMs in generating secure code fixes when provided with accurate contextual information. These results pave the way for further research and development in automating code security and improving the reliability of software systems.

1.5 Contributions

This thesis makes the following key contributions to the field of AI-assisted software security:

- **Novel Framework:** We propose a novel framework that integrates graph-based deep learning and LLMs for automated vulnerability detection and patching, bridging the gap between accurate identification and effective remediation.

- **Effective Vulnerability Detection and Patching:** We demonstrate the effectiveness of our approach in accurately detecting vulnerabilities, localizing them within code, and generating promising patches.

- **Importance of Prompt Engineering:** Our work emphasizes the critical role of carefully engineered prompts when using LLMs for security-critical tasks, showcasing how targeted guidance can significantly enhance the quality and security of generated code.

1.6 Thesis Organization

The remaining chapters of this thesis are organized as follows:

- Chapter 2: Background provides a comprehensive overview of essential concepts, including code vulnerabilities, CWE classifications, graph-based code representation (ASTs and CPGs), and Graph Neural Networks, with a particular emphasis on Graph Attention Networks (GATs).
- Chapter 3: Related Work surveys existing vulnerability detection and code repair techniques, comparing and contrasting various approaches.
- Chapter 4: Methodology presents details on dataset construction, model training, vulnerability localization and contextualization, and LLM-guided patching.
- Chapter 5: Implementation presents the implementation details of our framework, including the specific tools, libraries, and experimental configurations.
- Chapter 6: Evaluation and Results presents the findings of our experimental evaluation, analyzing the performance of our GAT model, the effectiveness of our localization technique, and the quality of the LLM-generated patches.
- Chapter 7: Conclusion summarizes our key findings, discusses limitations, and suggests directions for future research.

CHAPTER II

Background

This chapter lays the groundwork for understanding the key concepts and techniques employed throughout this thesis. We begin by delving into the nature of software vulnerabilities and their systematic categorization using frameworks like the Common Weakness Enumeration (CWE). We then explore the importance of representing code as graphs, focusing on Code Property Graphs (CPGs) as a powerful tool for capturing the intricate relationships between code elements. Finally, we introduce Graph Neural Networks (GNNs), specifically Graph Attention Networks (GATs), as a sophisticated deep learning technique well-suited for analyzing graph-structured data like CPGs, paving the way for more accurate and efficient vulnerability detection.

2.1 Understanding Code Vulnerabilities

Software vulnerabilities are weaknesses or flaws in code that deviate from secure coding practices, potentially enabling attackers to compromise system security. These flaws can manifest in various forms, often arising from complexities in software design, implementation errors, or a lack of awareness regarding secure coding principles. Understanding the nature and characteristics of these vulnerabilities is crucial for developing effective detection and remediation techniques.

Common Weakness Enumeration (CWE). To effectively address the diverse landscape of software vulnerabilities, systematic categorization is essential. The Common Weakness Enumeration (CWE) [16], maintained by MITRE, provides a widely-used framework for classifying software weaknesses based on their underlying nature and potential impact. CWE acts as a common language for security professionals, researchers, and developers, facilitating communication and collaboration in vulnerability analysis, mitigation, and prevention.

The CWE framework categorizes weaknesses based on several factors, including the affected software development lifecycle phase, the exploited weakness type (e.g., input validation, access control), and the potential impact of successful exploitation. This structured approach enables developers to identify common security flaws, prioritize remediation efforts based on risk assessments, and adopt secure coding practices that minimize the introduction of vulnerabilities.

This thesis focuses on five prevalent and impactful vulnerability types:

- **CWE-79: Cross-Site Scripting (XSS):** This vulnerability arises when web applications fail to properly sanitize user-supplied data, allowing attackers to inject malicious scripts into web pages viewed by other users [17]. This injection can lead to session hijacking, data theft, or the execution of arbitrary code in the victim's browser.

- **CWE-89: SQL Injection (SQLi):** SQLi vulnerabilities occur when user input used in constructing SQL queries is not properly sanitized [18]. This can allow attackers to manipulate the structure of the SQL query, potentially bypassing authentication mechanisms, accessing sensitive data, or even executing arbitrary commands on the database server.

- **CWE-384: Session Fixation:** This vulnerability arises when an attacker can manipulate a user's session ID, forcing it to a value known to the attacker. Once the user logs in, the attacker can hijack the session, gaining unauthorized access to the user's account [19].

- **CWE-22: Path Traversal:** This vulnerability stems from insufficient validation of user-supplied file paths, allowing attackers to access or manipulate files outside the intended directory [20]. Successful exploitation can grant attackers access to sensitive system files, configuration files, or enable arbitrary code execution.

- **CWE-78: OS Command Injection:** OS Command Injection vulnerabilities occur when applications allow user-supplied data to be executed as operating system commands without proper sanitization [21]. This can enable attackers to execute arbitrary commands on the underlying system, potentially leading to complete system compromise.

2.2 Code Property Graphs

While understanding the nature and categorization of software vulnerabilities is crucial, effectively detecting them within large and complex codebases requires sophisticated tools and techniques. Traditional code analysis methods often struggle to capture the intricate relationships and dependencies between different code elements, leading to incomplete or inaccurate vulnerability assessments. Code Property Graphs (CPGs) offer a powerful solution to this challenge by providing a unified and structured representation of code for comprehensive analysis.

Graph-Based Code Representation. Unlike traditional linear representations of code, which often focus on syntax and control flow, graph-based representations capture the essence of code as a network of interconnected entities and relationships. In this context, nodes typically represent program elements like variables, functions, classes, and statements, while edges depict various relationships between these elements, such as data flow, control flow, call relationships, and syntactic dependencies.

This graph-based approach provides a holistic view of the code, enabling more accurate and comprehensive vulnerability analysis.

CPGs: Combining Structure and Semantics. CPGs [13] build upon the foundation of graph-based code representation by integrating information from various program analyses, including Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs). This integration of syntactic and semantic information provides a rich and comprehensive representation of the code, enabling deeper insights into code behavior and vulnerability detection.

Leveraging ASTs for Syntactic Structure. ASTs capture the grammatical structure of the code, representing the hierarchical relationship between different code constructs. In the context of CPGs, AST information helps define the basic building blocks of the graph. Nodes representing variables, functions, and classes are derived from the AST, as are edges representing parent-child relationships between code blocks (e.g., a function definition containing multiple statements).

Incorporating CFGs for Control Flow Analysis. CFGs depict the possible execution paths within a program, showing how the program's control flow can branch based on conditional statements and loops. Integrating CFG information into the CPG allows for analyzing how data flows through different execution paths. This is particularly useful for detecting vulnerabilities like **Path Traversal (CWE-22)**, where an attacker manipulates the control flow to access files outside the intended directory, and **Improper Control Flow (CWE-201)**, where vulnerabilities arise from unexpected or manipulated program execution order.

Enhancing with DFGs for Data Flow Analysis. DFGs track how data propagates through the program, showing which variables influence other variables and how user input ultimately affects sensitive operations. Integrating DFG information into the CPG is crucial for identifying vulnerabilities related to data handling, such as **Cross-Site Scripting (XSS - CWE-79)**, where unsanitized user input flows into web page outputs, potentially enabling script injection attacks, and **SQL Injection (SQLi - CWE-89)**, where unsanitized user input used in SQL queries can allow attackers to manipulate database operations.

Properties: Adding Depth and Context. CPGs go beyond merely representing code elements and their relationships. They also incorporate properties associated with nodes and edges, providing additional context and information crucial for vulnerability analysis. These properties can include data types, indicating the type of data a variable can hold (e.g., integer, string, object); variable scopes, defining where a variable can be accessed within the code; function signatures, specifying the parameters a function accepts and the value it returns; and access modifiers, defining the accessibility of classes, methods, and variables (e.g., public, private, protected). These properties enhance the CPG's analytical capabilities by providing detailed information about each code element and relationship.

Advantages of CPGs for Vulnerability Detection. The comprehensive and unified nature of CPGs offers significant advantages for vulnerability detection, surpassing the limitations of traditional code analysis methods:

1. **Comprehensive Code Representation:** CPGs capture the intricate web of relationships within code, moving beyond simple syntactic analysis to enable a deeper understanding of code behavior and potential vulnerabilities.

2. **Facilitating Advanced Analyses:** The structured and graph-based format of CPGs makes them suitable for applying powerful graph algorithms and machine learning techniques to extract meaningful insights, identify patterns indicative of vulnerabilities, and automate vulnerability detection processes.

3. **Language Agnosticism:** While specific CPG construction techniques might differ between programming languages, the fundamental concept of representing code as a graph of entities and relationships remains applicable across a wide

range of languages, facilitating cross-language vulnerability analysis.

2.3 Graph Neural Networks for Code Representation

Traditional code analysis techniques often struggle to capture the complex, non-linear relationships present within software. This limitation has led to increasing interest in graph-based representations of code, enabling the application of powerful machine learning techniques like Graph Neural Networks (GNNs) for enhanced vulnerability detection.

Graph Neural Networks (GNNs) [22] excel at learning from graph-structured data. They operate through a process called message passing, where nodes iteratively exchange information with their neighbors. This allows each node to aggregate information from its surroundings, learning a vector representation (embedding) that encodes its local graph structure and features. These learned embeddings can then be used for various downstream tasks like node classification to identify vulnerable code snippets, graph classification to predict the presence of vulnerabilities within a program, and edge prediction to anticipate relationships between code elements.

Focus on Graph Attention Networks (GATs). While standard GNNs treat all neighboring nodes equally during message passing, Graph Attention Networks (GATs) [12] introduce a crucial advancement: an attention mechanism. This allows GATs to differentiate the importance of each neighboring node when aggregating information, similar to how humans focus on specific details when understanding a complex system.

This attention mechanism brings distinct advantages to code analysis. GATs excel at focusing on the most relevant connections in the code graph, such as a data flow edge connecting user input to a SQL query (critical for SQLi detection), while downplaying less informative connections, like a syntactic edge between unrelated variables. They also adapt well to the variable size and structure of code graphs by selectively attending to significant connections, potentially outperforming regular GNNs, which may struggle with the noise of less informative relationships in larger graphs.

GNNs for Vulnerability Detection. Recent research, such as the work by Zhou et al. [10], highlights the effectiveness of GNNs, particularly GATs, in achieving state-of-the-art accuracy in detecting code vulnerabilities. They have demonstrably outperformed traditional methods that rely on handcrafted features or shallower learning architectures.

2.4 Summary

This chapter established the foundational concepts and motivations for this thesis. We began by highlighting the critical need to address software vulnerabilities, particularly those identifiable through code structure analysis. We then introduced CWE classifications as a framework for categorizing these vulnerabilities and focusing our research on specific types.

CPGs emerged as a powerful approach for representing code in a manner that facilitates comprehensive analysis, capturing the intricate relationships between code elements often overlooked by traditional methods. We discussed the advantages of CPGs and the process of constructing them from source code.

Finally, we GNNs, specifically highlighting the strengths of GATs, as a potent deep learning technique for learning from graph-structured data. The combination of CPGs, with their rich representation of code, and GNNs, with their ability to learn complex patterns from graph data, sets the stage for developing more accurate and efficient automated vulnerability detection systems. The following chapters delve into related work, our proposed methodology, and the experimental evaluation of our approach, showcasing the potential of this synergy for enhancing software security.

CHAPTER III

Related Work

This chapter delves into the existing landscape of code vulnerability detection, surveying a range of approaches from traditional methods to cutting-edge applications of artificial intelligence (AI). We analyze their strengths, limitations, and how they relate to our proposed method of combining CPGs, GATs, and LLMs for automated vulnerability detection and patching.

3.1 Traditional Vulnerability Detection Techniques

Before the advent of AI, security researchers and practitioners primarily relied on manual code review and various automated but less sophisticated techniques. These methods, while still relevant in certain contexts, often face challenges in terms of scalability, accuracy, and their ability to cope with the ever-increasing complexity of modern software systems.

3.1.1 Manual Code Review

Manual code review involves the meticulous inspection of source code by security experts, who leverage their knowledge and experience to identify potential vulnerabilities. This process typically entails scrutinizing the code for insecure coding practices, logical flaws, potential security loopholes, and violations of established security guidelines. While manual code review remains a valuable approach for critical software systems, where accuracy and thoroughness are paramount, it suffers from inherent limitations that hinder its applicability to large-scale software projects.

The primary challenge lies in scalability. Reviewing large codebases manually is an incredibly time-consuming and resource-intensive endeavor. As software projects grow in size and complexity, the time required for comprehensive manual review becomes impractical, especially under tight development timelines [1]. Additionally, manual review is inherently subjective, as vulnerability assessments can vary between reviewers based on their experience, expertise, and understanding of the codebase [23]. This subjectivity can lead to inconsistencies in the identification and reporting of vulnerabilities, making it difficult to ensure a consistent level of security across a project. Furthermore, certain vulnerabilities stem from the complex interaction of multiple code components, making them exceptionally difficult to detect through manual inspection alone. A human reviewer might not readily grasp the intricate interplay of different code sections and their combined security implications [24]. This limitation becomes particularly pronounced in modern software architectures, which often involve distributed systems, microservices, and complex data flows, making it difficult for a single reviewer to track all potential vulnerabilities arising from the interaction of multiple components.

3.1.2 Static Analysis

Static analysis tools aim to automate the code review process by examining source code without executing it. These tools utilize various techniques to identify potential vulnerabilities, ranging from simple pattern matching to more sophisticated data flow and control flow analyses. Simple static analysis tools rely on pattern matching, searching for specific code structures or keywords known to be associated with vulnerabilities. For example, a tool might flag the use of dangerous functions like

”strcpy” in C, which is known to be susceptible to buffer overflow vulnerabilities. More sophisticated static analysis tools employ data flow analysis to track how data moves through the program and identify potential security issues. This involves tracing the flow of data from its source, such as user input, to sensitive operations, such as database queries or file system interactions, to detect vulnerabilities like SQL injection or cross-site scripting (XSS) [25].

Control flow analysis examines the possible execution paths within a program to identify potential security flaws. This analysis can detect vulnerabilities like path traversal or denial-of-service attacks, where an attacker can manipulate the control flow to gain unauthorized access or disrupt the application's functionality. Static analysis offers advantages in terms of efficiency and broad coverage compared to manual code review. These tools can analyze large codebases relatively quickly, making them suitable for integration into the software development lifecycle to provide continuous feedback to developers. However, static analysis tools also face limitations. They are often plagued by false positives, flagging code that is not actually vulnerable, which can lead to wasted effort in manual verification and potentially erode trust in the tool's results [3]. Many static analysis tools operate primarily on syntactic rules and predefined patterns, making them less effective at identifying vulnerabilities that arise from subtle interactions between code components or those requiring a deeper semantic understanding of the code's functionality [2]. They also struggle to handle code exhibiting dynamic behavior, relying heavily on external libraries, or using reflection, as it becomes challenging to reason about all possible execution paths and their potential security implications statically [26]. Despite these limitations, static analysis remains a widely used technique for vulnerability detection, with popular tools like SonarQube, Coverity, and Checkmarx offering varying degrees of sophistication and coverage, catering to different programming languages and development environments.

3.1.3 Fuzzing

Fuzzing, also known as fuzz testing, takes a dynamic approach to vulnerability detection by providing invalid, unexpected, or random data as input to a program, aiming to trigger crashes, errors, or unexpected behavior that might reveal vulnerabilities. This technique is particularly effective at uncovering certain types of vulnerabilities, especially those related to memory corruption and input validation. Fuzzing excels at detecting memory corruption vulnerabilities like buffer overflows, where an attacker can overflow a buffer with data to overwrite adjacent memory locations, potentially leading to arbitrary code execution. It can also be effective in identifying input validation vulnerabilities, where the application fails to properly validate user-supplied data, leading to potential security risks. By providing a wide range of unexpected or malformed inputs, fuzzing can uncover vulnerabilities that might not be apparent through manual code review or static analysis.

However, fuzzing also has limitations. Achieving comprehensive code coverage through fuzzing can be challenging. The effectiveness of fuzzing depends on the quality and diversity of the generated test cases, and it might not always reach all parts of the code, potentially missing vulnerabilities in less-tested areas [27]. Achieving comprehensive code coverage through fuzzing often requires significant computational resources and time, making it resource-intensive for complex software [5]. Additionally, fuzzing is less effective at uncovering logic-based vulnerabilities, such as those related to authentication bypasses, authorization flaws, or insecure business logic. These vulnerabilities often require a more nuanced understanding of the application's intended behavior and security requirements, which fuzzing, primarily focused on triggering crashes or errors, might not be able to capture [28]. Popular fuzzing tools include AFL (American Fuzzy Lop), which employs genetic algorithms to efficiently generate test cases, and LibFuzzer, an in-process, coverage-guided fuzzing engine integrated with the LLVM compiler infrastructure.

3.2 AI-Powered Vulnerability Detection

The limitations of traditional methods and the increasing complexity of modern software have fueled significant research into leveraging AI, particularly machine learning and deep learning, for vulnerability detection. These techniques aim to automate the process of vulnerability identification by learning from data and identifying patterns that might indicate security weaknesses.

3.2.1 Deep Learning for Vulnerability Detection

Deep learning models, with their ability to learn complex patterns and representations from large datasets, have shown promise in identifying vulnerabilities in code. These models can analyze code and learn to differentiate between secure and insecure coding patterns, potentially discovering vulnerabilities that might be missed by traditional rule-based methods. One approach treats code as natural language, applying Natural Language Processing (NLP) techniques to learn vulnerability patterns from code syntax and semantics. This approach leverages the success of NLP techniques in processing and understanding natural language text and applies them to the domain of code analysis. Models like Recurrent Neural Networks (RNNs) [7] and Transformers [29] have been used to analyze code as a sequence of tokens, similar to sentences, and learn to identify patterns and anomalies that might suggest vulnerabilities.

Another approach focuses on extracting handcrafted features from code, such as software metrics, control flow patterns, or data flow characteristics, and using these features to train classifiers like Support Vector Machines (SVMs) or deep neural networks. This approach relies on domain expertise to define features that capture relevant aspects of the code's structure and behavior for vulnerability detection. For example, researchers have used software metrics like cyclomatic complexity, which measures the number of independent paths through a program, to predict the likelihood of vulnerabilities [30]. Others have focused on extracting control flow patterns, such as the presence of loops or conditional statements, to identify potential vulnerabilities related to program logic [31]. Data flow analysis techniques have also been used to extract features related to how data flows through the program, such as the sources of data, the operations performed on the data, and the sinks where the data is ultimately used, to detect vulnerabilities like SQL injection or cross-site scripting [32].

Deep learning offers several potential advantages for vulnerability detection, including scalability, accuracy, and generalizability. These models can learn complex relationships and patterns from data, potentially identifying vulnerabilities that might be missed by rule-based approaches. Deep learning models can be trained on massive datasets of code, allowing them to learn from a wide range of coding practices and potentially identify vulnerabilities across diverse programming languages and software domains. Well-trained deep learning models can potentially generalize to new, unseen code, allowing them to identify vulnerabilities in code that was not part of the training data. However, they also face challenges related to their dependence on high-quality data and their inherent black-box nature. The performance of deep learning models heavily relies on the quality and diversity of the training data. Obtaining large, well-labeled datasets for security-related tasks can be challenging, as manually labeling vulnerabilities is time-consuming, requires specialized expertise, and might not always be feasible for certain types of vulnerabilities [33]. The decision-making process of deep learning models is often opaque, making it difficult to understand why a particular code snippet is flagged as vulnerable. This lack of interpretability can hinder trust in the model's predictions and make it challenging for developers to understand and fix the identified vulnerabilities [34].

3.2.2 Graph Neural Networks for Code Analysis

Recognizing the inherent graph-like nature of code, researchers are increasingly exploring the use of Graph Neural Networks (GNNs) for vulnerability detection. Unlike traditional deep learning models that often treat code as sequential data, GNNs are specifically designed to learn from data represented as graphs, making them well-suited for capturing the complex relationships and dependencies that exist within code [11]. GNNs operate on the principle of message passing, where nodes in the graph iteratively exchange information with their neighbors, allowing the model to learn a representation of each node based on its local neighborhood and the overall graph structure. This ability to capture both local and global information makes GNNs particularly effective in analyzing code, where vulnerabilities often arise from the interaction of multiple code elements and their relationships.

One promising area of research involves using Code Property Graphs (CPGs), with their rich semantic information, as input to GNNs. CPGs combine information from various program analyses, such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs), to create a comprehensive graph representation of the code. This representation captures not only the syntactic structure of the code but also its control flow, data flow, call relationships, and other dependencies, providing a rich context for vulnerability detection. GNNs trained on CPGs have shown encouraging results in identifying vulnerabilities that rely on understanding the flow of data and control within a program. For example, researchers have demonstrated the effectiveness of GNNs in detecting vulnerabilities like SQL injection and buffer overflows by analyzing the data flow paths within CPGs [10, 35]. Others have explored using GNNs to learn representations of function call graphs from CPGs to identify vulnerable functions based on their interactions with other functions [36].

GNNs offer several advantages for vulnerability detection, including their ability to model relationships, contextual awareness, and potential for interpretability. GNNs excel at capturing complex relationships within code, making them well-suited for identifying vulnerabilities that arise from the interaction of multiple code elements. They can learn representations of code that incorporate contextual information from the surrounding code, enabling them to identify vulnerabilities that might be missed by methods relying solely on local code patterns. While GNNs can be complex, they offer better interpretability compared to some other deep learning approaches, as it is often possible to analyze the attention weights or message passing mechanisms to understand which parts of the graph were most influential in the model's decision. However, GNNs also face challenges. Building and analyzing large CPGs can be computationally expensive, posing scalability challenges for analyzing very large codebases [37]. The complexity of GNN models themselves can also contribute to computational challenges, especially when dealing with deep GNN architectures or large graphs. While GNNs offer better interpretability compared to some other deep learning approaches, explaining their predictions can still be challenging, especially when dealing with complex graph structures and numerous features [38].

3.3 Leveraging LLMs for Code Remediation

Large Language Models (LLMs) like Codex [39] and GPT-3 [40] have gained significant attention for their impressive code generation capabilities. These models, trained on massive code datasets, can generate code in various programming languages, complete code snippets, translate natural language descriptions into functional code, and even refactor or optimize existing code. Recent research has begun exploring the potential of LLMs for automated code repair, including fixing security vulnerabilities [14, 41]. These approaches leverage the LLM's vast knowledge of coding practices and security vulnerabilities, acquired during training, to generate code patches that aim to address identified security issues.

However, relying solely on LLMs for security-critical tasks can be risky. Despite their impressive capabilities, LLMs face challenges in guaranteeing the security and contextual appropriateness of the generated code. LLMs are primarily trained to generate code that is syntactically correct and consistent with common coding patterns observed in the training data. However, this does not guarantee that the generated code is inherently secure. The LLM's training data might contain insecure examples, potentially leading to the propagation of vulnerabilities in the generated code [42]. Without sufficient context, LLMs might misinterpret the intent of the code or apply overly general fixes that are not appropriate for the specific situation. This can lead to the introduction of new vulnerabilities or the breaking of existing functionality [43]. Providing LLMs with the necessary context to understand the specific vulnerability and the surrounding code is crucial for generating secure and effective code patches.

3.3.1 Machine Learning and LLM-based Approaches for Code Patching

While LLMs alone have shown promise in generating code, leveraging machine learning and combining it with the capabilities of LLMs has yielded further advancements in code patching. One such approach involves using machine learning to guide the LLM's patch generation process. For instance, Tufano et al. [44] developed a technique that utilizes a sequence-to-sequence model to predict the location of a bug and then uses this information to guide an LLM in generating a patch. Another study by Dinella et al. [45] employed machine learning to rank candidate patches generated by an LLM, using features derived from the code and the vulnerability description. This ranking helps prioritize the most likely correct patches, improving the efficiency of the repair process.

Another direction involves training specialized machine learning models to generate patches directly. For example, Lutellier et al. [46] proposed using a transformer-based model to learn patch generation from a dataset of bug fixes, allowing the model to generate patches without relying on an LLM. While this approach offers potential advantages in terms of efficiency and control, it requires large, high-quality datasets of code patches for effective training.

Combining both machine learning and LLMs allows for leveraging the strengths of both approaches. Machine learning can provide guidance, ranking, or even direct patch generation, while LLMs can contribute their vast code knowledge and ability to generate syntactically and semantically coherent code.

3.4 Bridging the Gap: The Need for Our Approach

Our proposed methodology addresses the limitations of existing approaches by combining the strengths of CPGs, GATs, and LLMs in a novel way. We leverage the comprehensive and semantically rich representation provided by CPGs to capture the intricate relationships within code, enabling more accurate vulnerability detection by our GAT model. This approach exploits the power of graph neural networks to learn complex patterns and dependencies that characterize various vulnerability types, addressing the limitations of methods that rely on simplistic code patterns or fail to capture the broader context of the code.

Recognizing the risks associated with solely relying on LLMs for security-critical tasks, we utilize the output of our GAT-based vulnerability detection model, along with targeted code analysis, to provide the LLM with rich contextual information. This guidance enables the LLM to generate more secure and contextually appropriate code fixes, overcoming the LLM's limitations in understanding the specific vulnerability and its impact on the surrounding code. By combining the strengths of these techniques, we aim to create a more robust, accurate, and interpretable system for automated code vulnerability detection and patching. This hybrid approach seeks to exploit the advantages of each technique while mitigating their individual weaknesses, paving the way for more effective and reliable code security solutions.

CHAPTER IV

Proposed Methodology

This chapter details our novel methodology for automatically detecting and rectifying code vulnerabilities, leveraging a powerful synergy of graph-based deep learning and advanced language models. Recognizing the limitations of traditional vulnerability detection techniques, which often stop at identifying potential flaws, we aim to bridge the gap towards automated code remediation. Our approach harnesses the combined strengths of ASTs, CPGs, GATs, and prompt engineering to create a system that not only pinpoints security flaws but also guides the generation of targeted and context-aware code fixes.

4.1 Motivational Example: LLMs and Security Challenges

The following example highlights the limitations of relying solely on LLMs for secure code generation and motivates the need for our proposed framework.

Consider the prompt:

```
Create a Python Flask app that allows users to register, log in, and view
    their profile information stored in a SQLite database.
```

This seemingly innocuous prompt, devoid of any explicit security directives, led ChatGPT (as of the current model version) to generate the following Python code:

```python
from flask import Flask, render_template, request, redirect, url_for,
    session, g
import sqlite3
from werkzeug.security import generate_password_hash, check_password_hash

app = Flask(__name__)
app.secret_key = 'your_secret_key'
DATABASE = 'database.db'

def get_db():
    db = getattr(g, '_database', None)
    if db is None:
        db = g._database = sqlite3.connect(DATABASE)
    return db

@app.teardown_appcontext
def close_connection(exception):
    db = getattr(g, '_database', None)
    if db is not None:
        db.close()

def init_db():
    with app.app_context():
        db = get_db()
        cursor = db.cursor()
        cursor.execute('''CREATE TABLE IF NOT EXISTS users (
                          id INTEGER PRIMARY KEY AUTOINCREMENT,
                          username TEXT UNIQUE NOT NULL,
                          password TEXT NOT NULL
```

```python
                )''')
        db.commit()

@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        hashed_password = generate_password_hash(password, method='sha256')

        db = get_db()
        cursor = db.cursor()
        try:
            cursor.execute("INSERT INTO users (username, password) VALUES
                (?, ?)", (username, hashed_password))
            db.commit()
            return redirect(url_for('login'))
        except sqlite3.IntegrityError:
            return 'Username already exists'

    return render_template('register.html')

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        db = get_db()
```

```
        cursor = db.cursor()
        cursor.execute("SELECT * FROM users WHERE username = ?",
            (username,))
        user = cursor.fetchone()

        if user and check_password_hash(user[2], password):
            session['user_id'] = user[0]
            return redirect(url_for('profile'))
        else:
            return 'Invalid username or password'

    return render_template('login.html')

@app.route('/profile')
def profile():
    if 'user_id' not in session:
        return redirect(url_for('login'))

    db = get_db()
    cursor = db.cursor()
    cursor.execute("SELECT * FROM users WHERE id = ?",
        (session['user_id'],))
    user = cursor.fetchone()

    return render_template('profile.html', user=user)

@app.route('/logout')
def logout():
    session.pop('user_id', None)
```

```python
    return redirect(url_for('login'))


if __name__ == '__main__':
    init_db()
    app.run(debug=True)
```

This code, while functional, is susceptible to the following vulnerabilitis:

- **Cross-Site Scripting (XSS):** The code doesn't explicitly sanitize user input (username, password) before displaying it.
  - **Impact:** An attacker could potentially inject malicious JavaScript code into the forms or profile page, which would then execute in the browser of other users.
- **SQL Injection:** ChatGPT used parameterized SQL queries (cursor.execute("...", (variables,))), which helps mitigate SQL injection risks significantly.
- **Session Management:** The secret key (app.secret_key) is hardcoded directly into the script.
  - **Impact:** In a real-world scenario, this secret key should be stored more securely (environment variable or a separate configuration file) to prevent unauthorized access to session data.
- **Database Configuration:** The database name (DATABASE = 'database.db') is hardcoded.
  - **Impact:** Makes it slightly less flexible for deployment.

This experiment highlights the inherent security risks lurking within AI-generated code when security is not explicitly prioritized. As the adoption of AI-powered code generation proliferates, the urgency to address this security gap intensifies. The generated code lacks input sanitization, leaving it susceptible to Cross-Site Scripting (XSS) attacks, where malicious users could inject and execute arbitrary JavaScript code within the browsers of unsuspecting users.

Attempting to leverage the LLM's knowledge for remediation, we can provide the generated Flask app code back to ChatGPT with the following prompt:

```
    The following Python code implements a simple user registration and
        login system using Flask and SQLite. Please review the code and
        make any necessary changes to improve its security.


    from flask import Flask, render_template, request, redirect, url_for,
        session, g
import sqlite3
from werkzeug.security import generate_password_hash, check_password_hash


app = Flask(__name__)
app.secret_key = 'your_secret_key'
DATABASE = 'database.db'


def get_db():
    db = getattr(g, '_database', None)
    if db is None:
        db = g._database = sqlite3.connect(DATABASE)
    return db


@app.teardown_appcontext
def close_connection(exception):
    db = getattr(g, '_database', None)
    if db is not None:
```

```
        db.close()


def init_db():
    with app.app_context():
        db = get_db()
        cursor = db.cursor()
        cursor.execute('''CREATE TABLE IF NOT EXISTS users (
                            id INTEGER PRIMARY KEY AUTOINCREMENT,
                            username TEXT UNIQUE NOT NULL,
                            password TEXT NOT NULL
                        )''')
        db.commit()


@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        hashed_password = generate_password_hash(password, method='sha256')

        db = get_db()
        cursor = db.cursor()
        try:
            cursor.execute("INSERT INTO users (username, password) VALUES
                (?, ?)", (username, hashed_password))
            db.commit()
            return redirect(url_for('login'))
        except sqlite3.IntegrityError:
            return 'Username already exists'
```

```
    return render_template('register.html')


@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        db = get_db()
        cursor = db.cursor()
        cursor.execute("SELECT * FROM users WHERE username = ?",
            (username,))
        user = cursor.fetchone()

        if user and check_password_hash(user[2], password):
            session['user_id'] = user[0]
            return redirect(url_for('profile'))
        else:
            return 'Invalid username or password'

    return render_template('login.html')


@app.route('/profile')
def profile():
    if 'user_id' not in session:
        return redirect(url_for('login'))

    db = get_db()
```

```
    cursor = db.cursor()
    cursor.execute("SELECT * FROM users WHERE id = ?",
        (session['user_id'],))
    user = cursor.fetchone()

    return render_template('profile.html', user=user)

@app.route('/logout')
def logout():
    session.pop('user_id', None)
    return redirect(url_for('login'))

if __name__ == '__main__':
    init_db()
    app.run(debug=True)
```

The LLM returns the following code as a response to this prompt:

```
    from flask import Flask, render_template, request, redirect, url_for,
        session, g
import sqlite3
from werkzeug.security import generate_password_hash, check_password_hash
import os
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address
from itsdangerous import URLSafeTimedSerializer

app = Flask(__name__)
app.secret_key = os.environ.get('SECRET_KEY', 'default_secret_key')
DATABASE = 'database.db'
```

```
limiter = Limiter(
    get_remote_address,
    app=app,
    default_limits=["200 per day", "50 per hour"]
)

def get_db():
    db = getattr(g, '_database', None)
    if db is None:
        db = g._database = sqlite3.connect(DATABASE)
    return db

@app.teardown_appcontext
def close_connection(exception):
    db = getattr(g, '_database', None)
    if db is not None:
        db.close()

def init_db():
    with app.app_context():
        db = get_db()
        cursor = db.cursor()
        cursor.execute('''CREATE TABLE IF NOT EXISTS users (
                          id INTEGER PRIMARY KEY AUTOINCREMENT,
                          username TEXT UNIQUE NOT NULL,
                          password TEXT NOT NULL
                      )''')
        db.commit()
```

```python
@app.route('/register', methods=['GET', 'POST'])
@limiter.limit("5 per minute")
def register():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        # Input validation
        if not username or not password:
            return 'Username and password are required'
        if len(username) < 3 or len(password) < 6:
            return 'Username must be at least 3 characters and password at
                least 6 characters long'

        hashed_password = generate_password_hash(password,
            method='pbkdf2:sha256', salt_length=16)

        db = get_db()
        cursor = db.cursor()
        try:
            cursor.execute("INSERT INTO users (username, password) VALUES
                (?, ?)", (username, hashed_password))
            db.commit()
            return redirect(url_for('login'))
        except sqlite3.IntegrityError:
            return 'Username already exists'

    return render_template('register.html')
```

```
@app.route('/login', methods=['GET', 'POST'])
@limiter.limit("5 per minute")
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        db = get_db()
        cursor = db.cursor()
        cursor.execute("SELECT * FROM users WHERE username = ?",
            (username,))
        user = cursor.fetchone()

        if user and check_password_hash(user[2], password):
            session['user_id'] = user[0]
            session.permanent = True # Use permanent sessions
            return redirect(url_for('profile'))
        else:
            return 'Invalid username or password'

    return render_template('login.html')

@app.route('/profile')
def profile():
    if 'user_id' not in session:
        return redirect(url_for('login'))

    db = get_db()
    cursor = db.cursor()
```

```
    cursor.execute("SELECT * FROM users WHERE id = ?",
        (session['user_id'],))
    user = cursor.fetchone()

    return render_template('profile.html', user=user)

@app.route('/logout')
def logout():
    session.pop('user_id', None)
    return redirect(url_for('login'))

if __name__ == '__main__':
    init_db()
    app.run(debug=True, ssl_context='adhoc')
```

While ChatGPT responds with some generic security recommendations and implements minor changes, it fails to address the core XSS vulnerability. Following are the major takeaways from the generated response:

- **Superficial Fixes:**
  - While it correctly moved the secret key to an environment variable, it provided an insecure default ('default_secret_key') in the code, which defeats the purpose.
  - Using ssl_context='adhoc' for HTTPS is only suitable for development and is not recommended for production.
- **Generic Recommendations:** It provided several valid but generic security recommendations (SQL Injection, Session Fixation), without concrete implementation details specific to the given codebase.

- **Missed the Mark (XSS):** The most glaring omission is that ChatGPT completely failed to address the Cross-Site Scripting (XSS) vulnerability. It added input validation for length but didn't implement any output encoding/sanitization, leaving the application still vulnerable.

This example underscores a critical limitation of relying solely on LLMs for security-critical tasks. Without specific guidance on the nature and location of vulnerabilities within the code, LLMs tend to apply superficial fixes or suggest overly broad security measures, often failing to effectively mitigate the underlying risks. Our proposed framework addresses this limitation by combining the strengths of a vulnerability detection model (based on GATs) with the generative capabilities of LLMs, providing targeted guidance to ensure the generation of secure and effective patches.

4.2 Approach Overview

Our methodology unfolds in two distinct phases, as depicted in Figures 4.1 and 4.2. The training phase focuses on developing a robust vulnerability detection model using GATs, while the inference phase leverages this trained model for real-time vulnerability identification and guides an LLM for code patching.

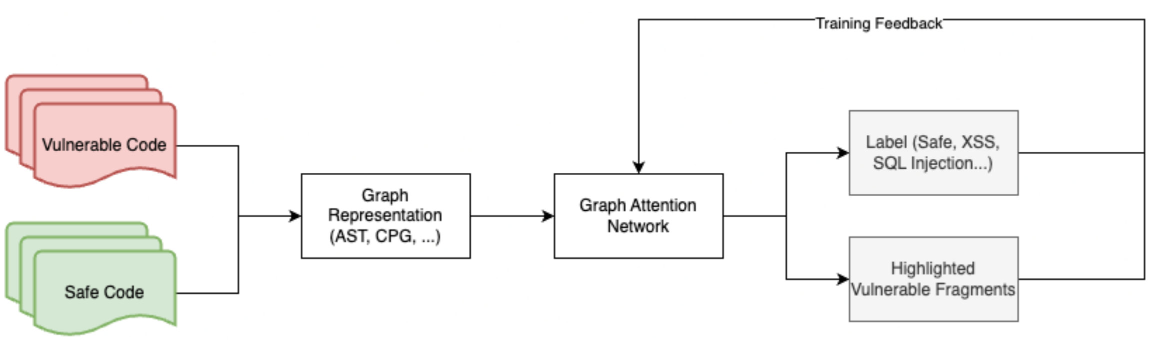

Figure 4.1: Training Pipeline: An illustration of the process of training our GAT model for vulnerability detection.

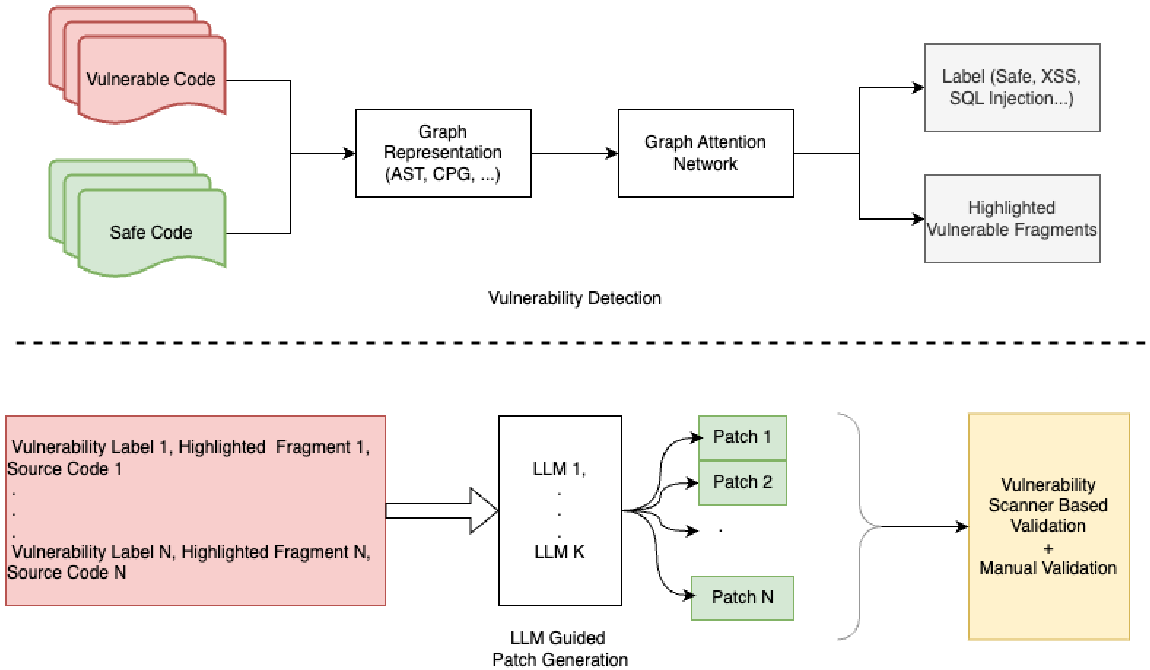

Figure 4.2: Inference Pipeline: Details of the steps involved in using our trained GAT model for real-time vulnerability identification and leveraging an LLM for code patching.

4.3 Phase 1: Training the Vulnerability Detection Model

The first phase of our methodology is dedicated to training a robust vulnerability detection model based on Graph Attention Networks (GATs). This phase involves carefully preparing a suitable dataset and leveraging the power of GATs to learn patterns indicative of vulnerabilities within code.

4.3.1 Dataset Preparation

A comprehensive and well-structured dataset is crucial for training an effective vulnerability detection model. Our dataset preparation methodology involves four key steps:

Diverse Data Acquisition. We gather Python code from diverse sources, including publicly available repositories like GitHub and curated vulnerability datasets such as Snyk's vulnerability database. Focusing on code relevant to web development and security ensures that our dataset captures a realistic range of coding practices and potential vulnerabilities.

Systematic Vulnerability Labeling. Accurate vulnerability labeling is essential for training a model that can effectively distinguish between secure and vulnerable code. Our labeling process combines three approaches:

- **Manual Code Review:** Security experts manually inspect a subset of the collected code snippets to identify and label vulnerabilities. This time-consuming but highly accurate approach provides a gold standard for evaluating the performance of our automated labeling methods.

- **Static Analysis** Tools: We utilize state-of-the-art static analysis tools like SonarQube [47], Bandit [15], and Semgrep [48] to automatically detect potential vulnerabilities within the code. These tools offer scalability and speed but require careful configuration and filtering of results to minimize false positives.

- **Pattern Recognition:** We develop and employ vulnerability-specific patterns and regular expressions to automatically identify code structures or function calls commonly associated with known vulnerabilities. This approach offers efficiency for detecting well-defined vulnerability types but might be limited in its scope.

Strategic Data Augmentation. Real-world codebases often exhibit biases in the distribution of vulnerability types. To ensure our model is trained on a diverse and balanced dataset, we augment our data by introducing synthetic vulnerabilities. This involves carefully crafting templates and using rule-based approaches to introduce variations of known vulnerabilities into existing code snippets, ensuring syntactic validity and semantic consistency.

Structured Dataset Organization. The final dataset is meticulously structured to facilitate efficient processing during model training and evaluation. Each data point consists of:

- **Code Snippet:** The raw Python code (e.g., a function or a block of code) being analyzed.
- **Vulnerability Label:** A binary label indicating whether the code snippet contains a vulnerability (1) or not (0).
- **Vulnerability Type:** If available, the specific CWE category of the vulnerability (e.g., CWE-89 for SQL injection) is included, allowing us to train models for multi-class vulnerability classification.
- **Metadata:** Additional information relevant to the code snippet, such as its source, the severity level of the vulnerability (if applicable), or the specific line numbers of the vulnerable code.

4.3.2 Vulnerability Detection Model Training Using GATs

Code Representation: ASTs and CPGs. We explore two primary graph representations of code to evaluate their impact on vulnerability detection performance:

- **Abstract Syntax Trees (ASTs):** ASTs [49] represent the grammatical structure of code, showing how different language constructs are nested within each other. In an AST, nodes typically represent language keywords, identifiers (e.g., variable names), and literals, while edges represent syntactic relationships between them.
- **Code Property Graphs (CPGs):** CPGs offer a richer, semantically-enriched representation by combining information from ASTs, Control Flow Graphs (CFGs) [50], and potentially other program analyses. This allows CPGs to capture a more comprehensive view of the code, including data flow, control flow, call relationships, and other dependencies.

Training Methodology. We utilize a multi-layer GAT architecture for vulnerability detection, as illustrated in Figure 4.1. The chosen graph representation (AST or CPG) of each code snippet is provided as input to the GAT. Each node in the graph is initialized with a feature vector encoding relevant information about the corresponding code element, such as its type, data type (for variables), and any associated literals.

The GAT's attention mechanism [51] allows it to learn which nodes and edges in the graph are most relevant for identifying vulnerabilities. During message passing, nodes selectively attend to their neighbors, assigning higher weights to connections that carry more information about the potential vulnerability.

We train the GAT to minimize the binary cross-entropy loss between its predicted vulnerability probabilities and the true labels from our dataset. This encourages the model to accurately distinguish between vulnerable and non-vulnerable code snippets. We use the Adam optimizer [52] to update the model's parameters during training, an effective optimization algorithm commonly used in deep learning due to its ability to handle sparse gradients and converge efficiently.

4.4 Phase 2: Inference and AI-Powered Code Patching

The second phase of our methodology focuses on using the trained GAT model for real-time vulnerability detection and leveraging an LLM for generating potential code fixes. This phase involves accurately localizing the vulnerability within the code, extracting relevant contextual information, and crafting carefully engineered prompts to guide the LLM in generating appropriate patches.

Vulnerability Localization. The attention mechanism within GATs provides valuable insights into the model's decision-making process, allowing us to pinpoint the vulnerable code section. After obtaining a vulnerability prediction from the GAT, we analyze the attention weights assigned to each node in the input graph during the model's forward pass. Nodes with higher attention weights are considered more influential in the model's decision, suggesting a higher likelihood of their involvement in the vulnerability. We identify a contiguous span of highly attentive nodes as the most likely location of the vulnerability. This span represents the section of code that the model focused on most when making its prediction.

Contextual Information Extraction. To provide the LLM with a comprehensive understanding of the identified vulnerability and the surrounding code, we extract relevant contextual information:

- **Variable Information:** We extract details about the variables used within the vulnerable span, including data types, data sources (e.g., user input, database queries, function calls), and data usage within the vulnerable span (e.g., in calculations, string concatenation, conditional statements).

- **Function Call Analysis:** We analyze the functions called within the vulnerable span, extracting information such as function names, arguments passed to the functions, and the data types and potential values returned by the functions.

- **Control Flow Paths:** If using CPGs, we leverage the control flow information to reconstruct the possible execution paths leading to the vulnerable code. This analysis can reveal potential entry points for malicious input or unexpected program states that might trigger the vulnerability.

- **Comments and Docstrings:** We extract nearby comments and docstrings as they can provide valuable developer insights into the purpose of the code, potential concerns, or intended functionality.

AI-Powered Code Patching with LLMs. We leverage the extracted contextual information and the GAT's vulnerability prediction to guide a Large Language Model (LLM) in generating potential code fixes. Our approach employs carefully crafted prompts to elicit effective and contextually relevant patches. The structure and content of the prompts are crucial for guiding the LLM's code generation process. Our prompts typically include:

- **Instruction:** A clear instruction to the LLM to "fix the security vulnerability" or "generate a secure version of this code."
- **Original Code Snippet:** The original code snippet provided to the system, including the vulnerable portion.
- **Highlighted Vulnerable Span:** We clearly mark or delimit the specific section of code identified as vulnerable by the GAT. This helps focus the LLM's attention on the problematic area.
- **Vulnerability Type:** The predicted CWE category of the vulnerability (if available) is included to provide the LLM with specific knowledge about the type of security flaw.
- **Contextual Information:** The extracted contextual information is presented to the LLM in a structured and concise manner, allowing it to understand the surrounding code and the potential impact of the vulnerability.

We utilize a powerful pre-trained LLM, such as Codex [39] or GPT-3 [40], for code generation. The LLM, drawing upon its extensive knowledge of coding practices, security best practices, and the context provided in the prompt, generates candidate code patches to address the identified vulnerability. These generated patches are then subject to automated testing, static analysis, and potentially manual review to evaluate their quality and security, ensuring that they effectively remediate the vulnerability without introducing new issues.

4.5 Summary

This chapter has presented a detailed methodology for automatically detecting and rectifying code vulnerabilities. By combining the power of graph-based deep learning, specifically GATs, for precise vulnerability identification with the generative capabilities of LLMs for patch generation, our approach offers a promising avenue for enhancing software security.

CHAPTER V

Implementation

This chapter details the implementation of our proposed framework for automated vulnerability detection and patching. We present a detailed account of the techniques, tools, and resources used to construct our dataset, train our GAT model, and integrate an LLM for generating code fixes guided by the insights from our vulnerability analysis.

5.1 Dataset Construction

The foundation of any successful machine learning system lies in a well-constructed, representative dataset. Recognizing the scarcity of large-scale, labeled datasets specifically designed for graph-based code vulnerability analysis, we undertook a systematic approach to dataset creation, combining real-world code, established vulnerability patterns, and synthetic data generation.

Data Acquisition and Sources. We initiated our dataset construction by gathering a diverse corpus of Python code from two primary sources:

- **GitHub Repositories:** We collected Python code from publicly available repositories on GitHub, targeting projects related to web development and security. We focused our search on repositories tagged with keywords such as "web development," "flask," "django," and "security." This strategic selection ensured the inclusion of code likely to contain common web application vulnerabilities.

- **Software Assurance Reference Dataset (SARD):** While SARD [53] primarily focuses on C/C++, its wealth of vulnerability patterns and code structures provided valuable insights. We analyzed relevant vulnerability types within SARD and adapted these patterns to generate synthetic Python vulnerability examples, augmenting the diversity of our dataset.

This combined approach yielded an initial dataset of 4000 real-world Python files, which served as the foundation for vulnerability labeling, snippet extraction, and subsequent data augmentation.

Vulnerability Labeling. Accurate and comprehensive vulnerability labeling is paramount for training a model that can effectively distinguish between secure and insecure code. We adopted a multi-pronged strategy to achieve this:

- **Manual Code Review:** A subset of the GitHub-sourced Python files underwent meticulous manual inspection by security experts to identify and label vulnerabilities. The focus was on detecting instances of common web application vulnerabilities:
  - **SQL Injection (CWE-89):** Code vulnerable to manipulation of database queries through malicious user input [54].
  - **Cross-Site Scripting (XSS) (CWE-79):** Code allowing attackers to inject client-side scripts into web pages viewed by other users [55].
  - **Command Injection (CWE-78):** Code that allows attackers to execute arbitrary system commands [56].
  - **Path Traversal (CWE-22):** Code that allows attackers to access files or directories outside of the intended web root [57].

- **Static Analysis Tool Assistance:** We leveraged the Bandit static analysis tool [15], specifically designed for Python code, to assist in vulnerability identification. Bandit's findings were manually verified to ensure accuracy and relevance, minimizing the inclusion of false positives in our dataset.

- **Synthetic Vulnerability Injection:** To ensure sufficient representation of specific vulnerability types and address potential biases in the distribution of vulnerabilities in real-world code, we synthetically injected vulnerabilities into initially benign Python code snippets. We used carefully crafted templates, based on common vulnerability patterns and insecure coding practices, to introduce vulnerabilities in a controlled and syntactically valid manner.

Code Snippet Extraction. To facilitate efficient processing and focus our models on relevant code segments, we extracted self-contained code snippets from both the labeled GitHub-sourced files and the synthetically generated vulnerable examples. Each snippet, typically representing a function, a class, or a cohesive block of related code, was treated as an individual data point for subsequent analysis and model training.

Dataset Structure and Statistics. The final dataset consists of a diverse collection of 16,000 Python code snippets, meticulously labeled and categorized. Each data point includes the following information:

- **Code Snippet:** The raw Python code being analyzed.

- **Vulnerability Label and Type:** A label indicating the presence or absence of a vulnerability and the specific CWE category of the vulnerability. For vulnerable snippets, we included the CWE ID (e.g., CWE-89 for SQL Injection).

To address the potential for model bias towards vulnerable data, we included a substantial number of benign (non-vulnerable) code snippets. The final dataset composition is detailed in Table 5.1.

Table 5.1: Dataset Composition

Vulnerability Type	CWE ID	Count
SQL Injection	CWE-89	2000
Cross-Site Scripting (XSS)	CWE-79	2000
Command Injection	CWE-78	2000
Path Traversal	CWE-22	2000
Insecure Session Management	CWE-384	2000
Benign (Non-Vulnerable)	N/A	6000
Total		**16,000**

5.2 Code Representation: ASTs and CPGs

Our approach leverages two primary graph-based representations of code: Abstract Syntax Trees (ASTs) and Code Property Graphs (CPGs). Both representations provide structured views of the code, but they differ significantly in their level of semantic richness, impacting their suitability for training machine learning models for vulnerability detection. To illustrate these differences, we will examine a Python code snippet exhibiting a potential path traversal vulnerability.

```
import os

def handle_file_upload(filename):

    base_dir = "/var/www/uploads/"

    filepath = os.path.join(base_dir, filename)


    if ".." in filename: # Basic attempt to prevent path traversal

        return "Invalid filename."


    with open(filepath, "wb") as f:

        pass

        # ... (Code to write file data to disk)


    return f"File uploaded to: {filepath}"
```

5.2.1 Abstract Syntax Trees (ASTs)

ASTs represent the grammatical structure of a program, essentially capturing the code's parse tree. We used the Python `ast` module [58] to parse each code snippet and generate its corresponding AST. Each node in the AST corresponds to a language element, such as a function definition (def handle_file_upload...), a variable assignment (base_dir = "/var/www/uploads/"), a function call (os.path.join(base_dir, filename)), or a control flow statement (`if ".." in filename:`). Edges in the AST represent the syntactic relationships between these elements, primarily parent-child relationships that reflect the nesting of code constructs. Figure 5.1 visualizes the AST for our example code snippet.

While ASTs provide a fundamental representation of the code's structure, their focus on syntax limits their ability to capture the semantic nuances critical for understanding vulnerabilities. ASTs excel at representing the hierarchical organization of code elements and their syntactic roles but often fall short in conveying the flow of data and control that underpins many security weaknesses.

5.2.2 Code Property Graphs (CPGs)

CPGs extend ASTs by incorporating additional semantic information derived from various program analyses, enriching the representation with a deeper understanding of the code's behavior. We used the Joern tool [59], a robust open-source platform for code analysis, to construct CPGs from our Python code snippets. Joern performs a series of analyses, including:

- **Control Flow Analysis:** Determines the possible execution paths within the code, capturing how control is transferred between different statements and functions.

- **Data Flow Analysis:** Tracks the flow of data through the program, identifying how variables are defined, used, and modified, revealing potential paths for data manipulation and vulnerabilities.

- **Points-To Analysis:** Determines which memory locations a pointer variable might point to, essential for understanding pointer-related vulnerabilities in languages like C/C++.

By integrating the results of these analyses, CPGs capture a more comprehensive view of the code compared to ASTs. They include information about the flow of data through the program, the possible execution paths, the relationships between variables and functions, and potential points of vulnerability. Figure 5.2 depicts the CPG for the same code snippet.

5.2.3 Illustrative Example: AST vs. CPG

A comparison of Figures 5.1 and 5.2 highlights the key distinctions between ASTs and CPGs and underscores the advantages of CPGs for vulnerability detection.

The AST, in Figure 5.1, primarily focuses on the syntactic structure of the code. It accurately represents the function definition, variable assignments, conditional statement, and function calls. However, it lacks the semantic context to recognize the potential flow of user input (`filename`) into the sensitive file operation (`open`), which constitutes the path traversal vulnerability. The AST, by itself, cannot discern that the `filename` variable, potentially controlled by a malicious user, influences the `filepath` variable used in the `open` function, leaving the vulnerability undetected.

The CPG, depicted in Figure 5.2, offers a richer and more revealing representation. In addition to the syntactic structure captured by the AST, it includes data flow edges that explicitly show the movement of data through the code. These edges visually demonstrate how the `filename` variable, passed as input to the handle_file_upload function, flows through the `os.path.join` function and ultimately influences the filepath variable used in the **open** function. This visual representation clearly highlights how a malicious filename could potentially be used to access files outside the intended directory.

This additional information embedded within the CPG is crucial for vulnerability detection. It provides the necessary context for understanding how user input might be manipulated to exploit vulnerabilities, enabling machine learning models to learn more nuanced patterns and dependencies within the code. The richer semantic information captured by CPGs makes them a more suitable representation for training effective vulnerability detection models, enabling them to identify a wider range of vulnerabilities, including those that rely on understanding data flow and control flow relationships.

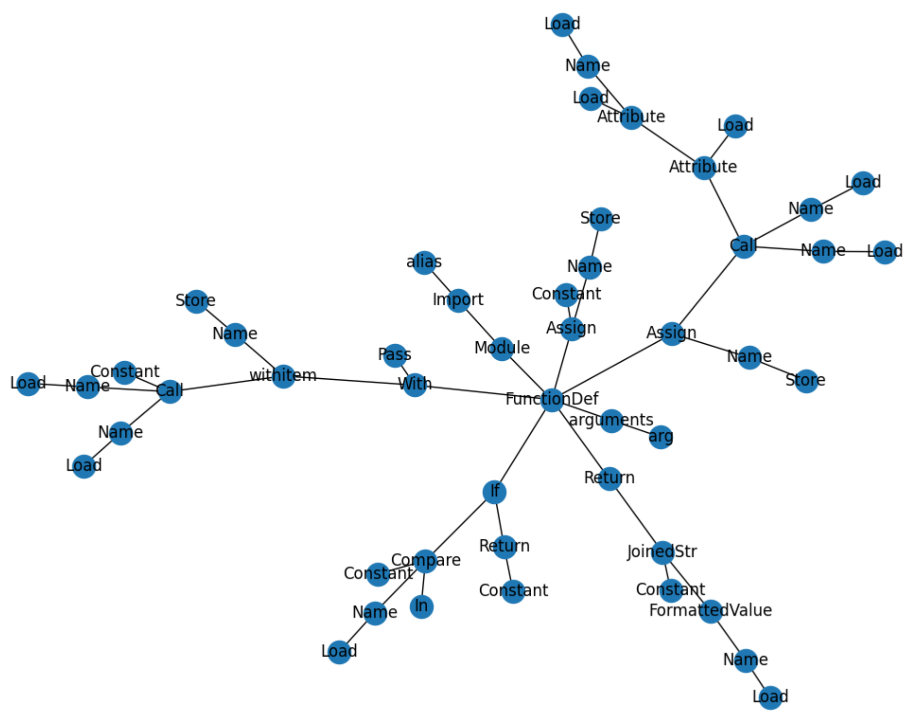

Figure 5.1: AST Representation for a Code Snippet

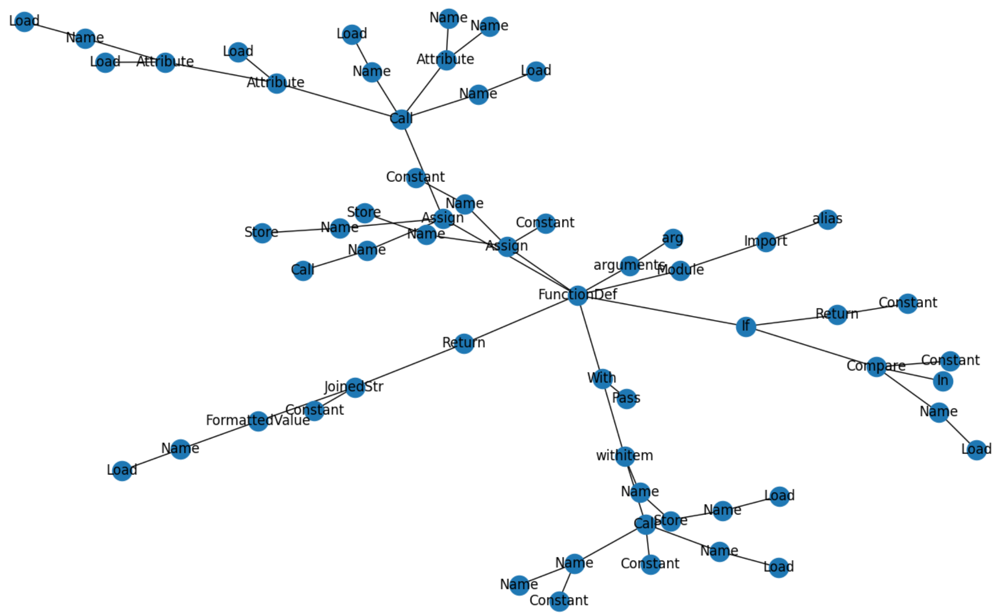

Figure 5.2: CPG Representation for the Same Code Snippet

In our framework, we initially experimented with both AST and CPG representations. However, our empirical evaluation confirmed that the GAT model trained on CPGs consistently outperformed the model trained on ASTs, showcasing the significance of incorporating semantic information for effective vulnerability detection. Therefore, we selected the CPG representation for all subsequent experiments and evaluations.

5.3 Feature Encoding

To enable our machine learning models to effectively learn from the AST and CPG representations, we encoded relevant information about each node in the graph as a feature vector. The specific features used for both representations are detailed below.

5.3.1 AST Feature Encoding

For each node in the AST, we extracted the following features:

- **Node Type:** The grammatical category of the node (e.g., `FunctionDef, Assign, Call, Name, Constant`). This feature captures the syntactic role of the node within the code.
- **Data Type:** For variables, the inferred data type (e.g., `str, int, float`). This feature provides information about the kind of data stored in the variable.
- **Literal Value:** For constant nodes, the actual literal value (e.g., `"user input", 10, 3.14`). This feature captures the value associated with the constant.

5.3.2 CPG Feature Encoding

For each node in the CPG, we extracted a richer set of features, leveraging the additional semantic information available in the CPG:

- **Node Type:** The type of the node (e.g., "Identifier," "Call," "Literal"). This feature categorizes the node based on its role in the code.
- **Code:** The actual code associated with the node (e.g., "os.path.join," "open"). This feature provides a more specific representation of the code element.
- **Data Type:** The data type associated with the node (e.g., "String," "Integer"). This feature provides information about the kind of data the node represents.
- **User Input Flag:** A boolean flag indicating whether the node is part of a function handling user input. This feature helps the model identify potential entry points for malicious data.

5.4 Model Architecture and Training

5.4.1 GAT Model Architecture

Our vulnerability detection model is based on Graph Attention Networks (GATs) [12], a powerful type of graph neural network designed to learn from graph-structured data. GATs incorporate an attention mechanism that allows them to selectively attend to different parts of the graph, learning which nodes and edges are most relevant for the task at hand. This makes GATs particularly well-suited for code analysis, where vulnerabilities often arise from the interaction of multiple code elements and their relationships.

We implemented our GAT model using the PyTorch Geometric library [60]. Our model architecture consists of two GAT layers stacked sequentially, with each layer employing eight attention heads. Each GAT layer maps the input features to a hidden dimension of 64. ReLU activation is applied after each layer to introduce non-linearity into the model.

To mitigate overfitting, we incorporated dropout with a rate of 0.5 after each GAT layer. Dropout randomly sets a fraction of the neuron activations to zero during training, forcing the model to learn more robust and generalizable representations.

5.4.2 Model Training

The training process involved splitting the dataset into training (70%), validation (15%), and testing sets (15%) using stratified sampling to ensure an even distribution of vulnerability types across the splits. This stratification ensures that the model is exposed to a representative sample of each vulnerability type during training and evaluation.

We utilized the Adam optimizer [52] for training our GAT model. Adam is a popular optimization algorithm known for its effectiveness in training deep learning models. We used a learning rate of $1e^{-4}$ and a weight decay of $1e^{-3}$ to regularize the model parameters and prevent overfitting.

The model was trained for a maximum of 50 epochs with a batch size of 16. We incorporated early stopping based on the validation loss to prevent overfitting. If the validation loss did not improve for a predefined number of epochs (patience), the training process was stopped to prevent the model from memorizing the training data and losing its ability to generalize to unseen examples.

5.4.3 Loss Function: Weighted Cross Entropy

We observed that our initial models exhibited a bias towards predicting vulnerabilities due to the class imbalance in the dataset, with a higher number of vulnerable samples compared to benign ones. To address this, we implemented a weighted cross-entropy loss function. Weighted cross entropy assigns higher weights to the minority classes, ensuring that the model is penalized more for misclassifying vulnerable code snippets. This weighting strategy helps the model learn to focus on detecting vulnerabilities more effectively, even when they are less frequent in the training data.

5.5 Vulnerability Localization and Contextualization

Once a code snippet is classified as potentially vulnerable, our framework proceeds to localize the vulnerability within the code and extract relevant contextual information to guide the LLM in generating appropriate code fixes.

5.5.1 Attention-Based Localization

We utilize the attention weights learned by our GAT model during training to guide the process of vulnerability localization. The attention mechanism within GATs assigns weights to different nodes and edges in the graph, reflecting their importance in the model's prediction. After obtaining a vulnerability prediction from the GAT model, we extract the attention weights assigned to each node in the input AST or CPG during the model's forward pass. Nodes with higher attention weights indicate greater influence on the model's prediction, suggesting a higher likelihood of their involvement in the vulnerability.

Rather than relying solely on the top-ranked node, we identify a contiguous span of highly attentive nodes as the most likely location of the vulnerability. This reflects the understanding that vulnerabilities often involve interactions between multiple code elements rather than a single isolated node. We empirically determined a threshold, selecting the top 5% of nodes with the highest attention weights to form the vulnerable span. This span represents the section of code that the model focused on most when making its vulnerability prediction.

5.5.2 Contextual Information Extraction

We developed a dedicated Python module for contextual information extraction. This module utilizes the `ast` library for AST parsing and a CPG parsing library, such as Joern [59], for CPG manipulation. We employed the visitor pattern [61] to efficiently traverse both AST and CPG representations. The visitor pattern allows us to define specific actions to be performed when encountering different node types during the traversal, enabling targeted extraction of relevant information.

The module extracts the following types of contextual information:

- **Variable Information:** We extract details about the variables used within the vulnerable span and its surrounding context. This information includes:
  - Variable names
  - Inferred data types from assignments
  - Literal values assigned to the variables
  - The scope of the variable (e.g., local, global)

  - Data flow relationships, if available in the CPG, showing how the variable's value propagates through the code

- **Function Call Analysis:** We analyze function calls within the vulnerable span, extracting information such as:
  - The name of the called function
  - Arguments passed to the function
  - Inferred return type of the function
  - If available in the CPG, the definition of the called function to provide further context about its behavior

- **Control Flow Reconstruction:** Utilizing the control flow information captured in the CPG, we reconstruct possible execution paths leading to the vulnerable code. This analysis reveals:
  - Potential entry points for malicious input, helping to understand how an attacker might exploit the vulnerability.
  - Conditions that might lead to the execution of the vulnerable code, providing insights into the circumstances under which the vulnerability might manifest.

- **Comment Extraction:** We extract nearby comments and docstrings that might provide valuable developer insights into the purpose of the code, potential concerns, or intended functionality. This information can be helpful for understanding the developer's intent and guiding the LLM in generating more appropriate code fixes.

5.6 AI-Powered Code Patching with LLMs

We leverage the extracted contextual information, along with the GAT model's vulnerability prediction and localization, to guide a Large Language Model (LLM) in generating potential code fixes. We employed Google Gemini Pro [62], a powerful LLM accessible through the Google AI Platform, for our code generation tasks.

5.6.1 Prompt Engineering

The design and structure of the prompts provided to the LLM are crucial for eliciting effective and contextually relevant code patches. Our prompts are carefully structured to provide the LLM with a comprehensive understanding of the vulnerability and the surrounding code, enabling it to generate targeted and appropriate fixes. The general structure of our LLM prompts is as follows:

```
### Task: Fix a security vulnerability in the following Python code.

The code has been identified as potentially containing a **[Vulnerability
    Type]** vulnerability.
The vulnerable part of the code is highlighted below:

```python
[Highlighted Vulnerable Code Segment]

Please provide a corrected version of the entire code that addresses this
    vulnerability while maintaining the original functionality:

[Original Code Snippet]

Important:
```
```

```
- Focus on fixing the specific vulnerability identified.
- Ensure the patched code is functionally equivalent to the original code.
- If the vulnerability cannot be fixed without more context, please
  explain why.
```

We replace the placeholders in the prompt template with the following information:

- `[Vulnerability Type]`: The specific CWE category of the vulnerability predicted by our GAT model.
- `[Highlighted Vulnerable Code Segment]`: The section of code identified as vulnerable by the GAT model, based on the attention weight analysis. This helps focus the LLM's attention on the problematic area.
- [**Original Code Snippet**]**:** The entire code snippet submitted for analysis.

5.6.2 Patch Generation and Evaluation

We provide the generated prompts to the Google Gemini Pro model, which then produces candidate code patches. The LLM, drawing upon its vast knowledge of coding practices and security best practices acquired during training, attempts to generate patches that address the identified vulnerability while preserving the functionality of the original code.

The generated patches are then evaluated for both correctness and security using a multi-faceted approach:

- **Automated Testing:** We develop test cases tailored to the specific functionality of the code snippet to assess the functional correctness of the patched code. This ensures that the original behavior of the code is preserved after the patch is applied.

- **Static Analysis:** We re-run the Bandit static analysis tool [15] on the patched code to check for the following:
  - The presence of the original vulnerability: This verifies whether the generated patch effectively addresses the identified security issue.
  - The introduction of new vulnerabilities: This ensures that the patch itself does not inadvertently introduce new security risks into the code.
- **Manual Review:** In some cases, especially for complex or subtle vulnerabilities, we perform manual code review to assess the quality and appropriateness of the generated patches. This involves expert scrutiny of the patched code to evaluate its security and ensure that it adheres to best practices.

5.7 Summary

This chapter detailed the implementation of our framework for automated vulnerability detection and patching. We described our methodology for constructing a comprehensive dataset, training a GAT model to identify vulnerabilities, localizing vulnerabilities within the code, extracting contextual information, and leveraging an LLM to generate potential code fixes. The following chapter will evaluate the performance of our framework, demonstrating its effectiveness in accurately identifying and addressing various vulnerability types.

CHAPTER VI

Evaluation and Results

This chapter presents a comprehensive evaluation of our proposed framework for automated vulnerability detection and patching. We first compare the performance of our Graph Attention Network (GAT) model trained on two distinct code representations: Abstract Syntax Trees (ASTs) and Code Property Graphs (CPGs). Following the selection of the optimal representation, we delve into a detailed assessment of the model's performance on the held-out test set, analyze the effectiveness of our attention-based localization technique, and critically examine the quality of LLM-generated patches for code remediation. Our evaluation utilizes multiple perspectives—human evaluation, static analysis with Bandit, and re-evaluation with our trained GAT model—providing a robust assessment of our framework's efficacy and its potential for real-world application.

6.1 Vulnerability Detection Performance: AST vs. CPG

We conducted initial experiments to compare the performance of our GAT model trained on two different code representations: ASTs and CPGs. Our goal was to determine which representation yielded better vulnerability detection accuracy, as the choice of code representation can significantly influence the model's ability to learn relevant patterns and dependencies within the code.

6.1.1 Evaluation Metrics

To evaluate the performance of our GAT models, we employed standard metrics for binary classification:

- **Accuracy:** Measures the overall correctness of the model's predictions, representing the proportion of correctly classified code snippets (both vulnerable and non-vulnerable) out of the total number of snippets.

- **Precision:** Focuses on the accuracy of positive predictions, quantifying the proportion of correctly identified vulnerable snippets out of all snippets classified as vulnerable. A high precision indicates a low rate of false positives.

- **Recall:** Measures the model's ability to identify all actual vulnerabilities, representing the proportion of correctly identified vulnerable snippets out of all actual vulnerable snippets in the dataset. A high recall signifies a low rate of false negatives.

- **F1-Score:** The harmonic mean of precision and recall, providing a balanced measure of the model's performance, particularly useful when dealing with imbalanced datasets.

These metrics provide a comprehensive view of the model's performance, considering both its ability to correctly identify vulnerable code and its ability to avoid misclassifying benign code as vulnerable.

6.1.2 Results and Analysis

Tables 6.1 and 6.2 summarize the performance of our GAT model on the validation set, trained separately on AST and CPG representations. As hypothesized, the GAT model trained on the CPG representation consistently outperformed the model trained on the AST representation across all evaluation metrics.

Metric	AST Representation
Accuracy	0.74
Precision	0.74
Recall	0.76
F1-Score	0.77

Table 6.1: GAT Performance on AST Representation

Metric	CPG Representation
Accuracy	0.86
Precision	0.81
Recall	0.85
F1-Score	0.86

Table 6.2: GAT Performance on CPG Representation

The CPG-based model demonstrated significantly higher accuracy, precision, recall, and F1-score compared to the AST-based model. This indicates its superior ability to accurately distinguish between vulnerable and non-vulnerable code snippets and to identify a larger proportion of actual vulnerabilities. The improvement in recall is particularly noteworthy, suggesting that the CPG-based model is more effective at finding vulnerabilities while minimizing false negatives. This finding strongly supports our hypothesis that the richer semantic information encoded in CPGs, including data flow and control flow relationships, provides a more informative context for vulnerability detection, leading to a more accurate and effective model.

The confusion matrices, depicted in Figures 6.1 and 6.2, offer a detailed breakdown of the model's performance across different vulnerability types. These matrices visualize the counts of true positives, true negatives, false positives, and false negatives for each vulnerability category.

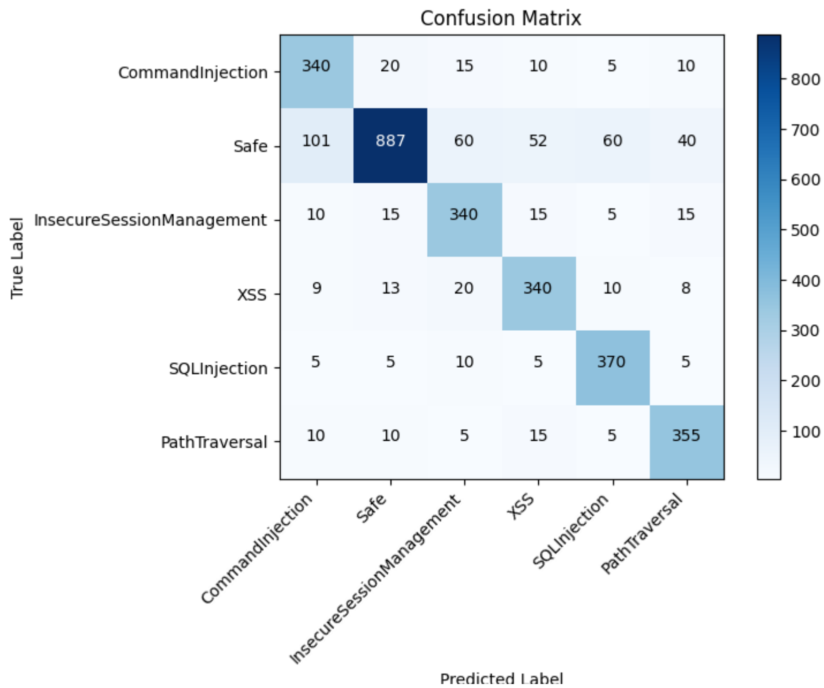

Figure 6.1: Confusion Matrix for GAT Model Trained on AST Representation

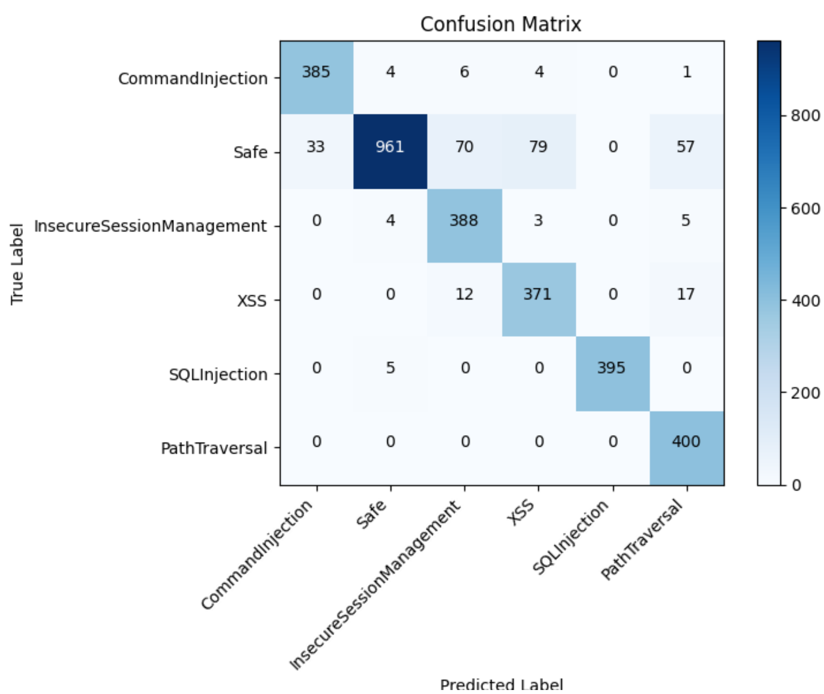

Figure 6.2: Confusion Matrix for GAT Model Trained on CPG Representation

Figures 6.3 and 6.4 illustrate the training loss and accuracy, respectively, over epochs for the AST-based model. These graphs provide insights into the training process and highlight the convergence behavior of the model. Similarly, Figures 6.5 and 6.6 depict the training loss and accuracy over epochs for the CPG-based model.

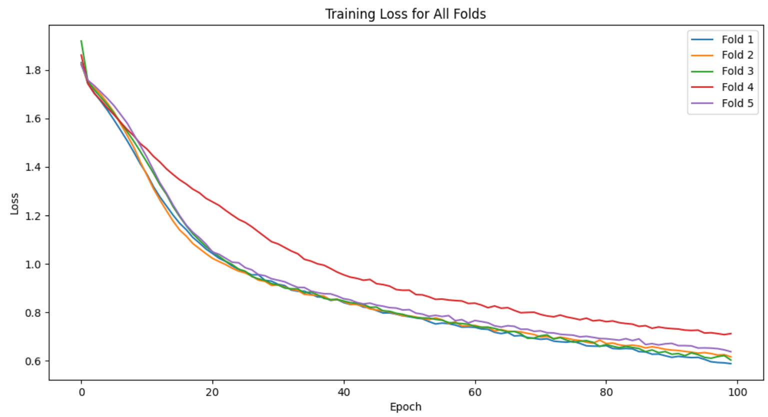

Figure 6.3: Training Loss for GAT Model Trained on AST Representation

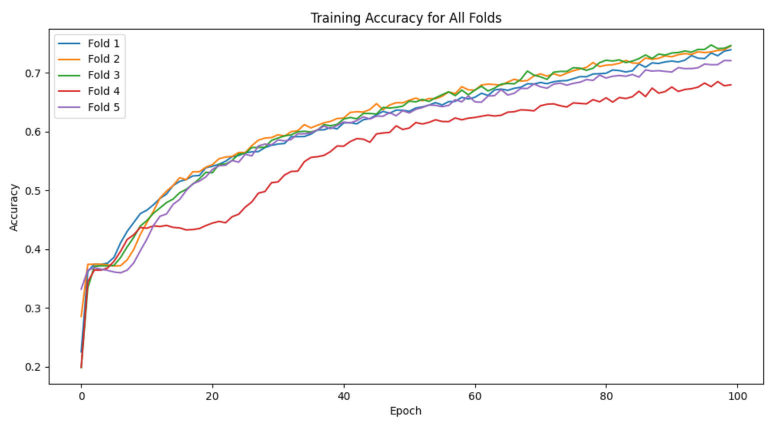

Figure 6.4: Training Accuracy for GAT Model Trained on AST Representation

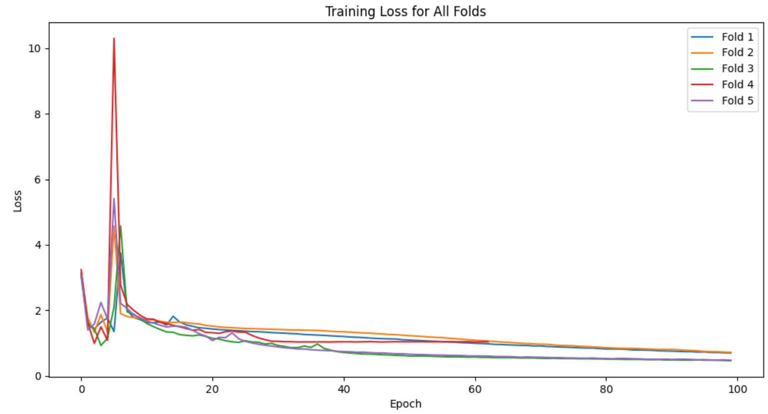

Figure 6.5: Training Loss for GAT Model Trained on CPG Representation

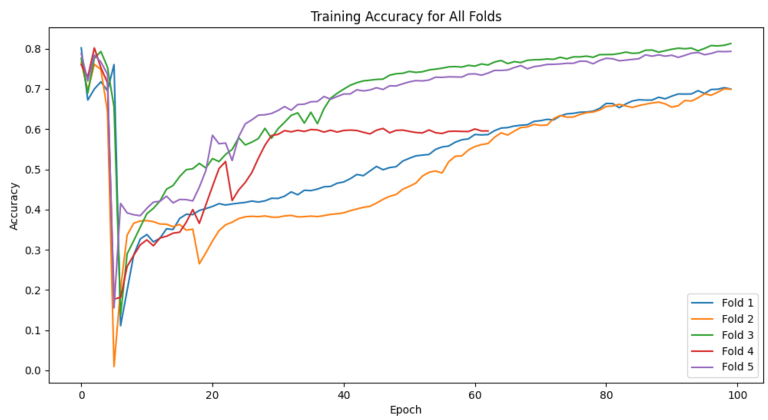

Figure 6.6: Training Accuracy for GAT Model Trained on CPG Representation

Based on the superior performance of the GAT model trained on the CPG representation, we selected the CPG representation for all subsequent experiments and evaluations. The richer semantic information embedded within CPGs proved crucial for achieving higher accuracy and recall in vulnerability detection.

6.2 Vulnerability Localization Effectiveness

Having established the superiority of the CPG representation, we proceeded to evaluate the effectiveness of our attention-based localization technique in pinpointing the vulnerable code sections within the snippets correctly classified as vulnerable by our GAT model. Accurate localization is crucial for guiding the LLM in generating targeted and effective code fixes.

6.2.1 Evaluation Approach

We manually examined a subset of 200 code snippets randomly selected from the test set where the GAT model, trained on the CPG representation, correctly predicted the presence of a vulnerability. For each snippet, we compared the vulnerable span identified by our attention-based method to the ground truth vulnerable lines of code, which were determined during the manual labeling process. This involved visually inspecting the highlighted code sections and comparing them to the actual lines of code known to contain the vulnerability.

6.2.2 Results and Analysis

Our attention-based localization technique achieved an accuracy of 87%, correctly identifying the vulnerable code span in 174 out of the 200 analyzed snippets. In the remaining 13% of cases, the highlighted span either included additional lines of code beyond the actual vulnerable section or missed a few lines within the vulnerable section. These discrepancies can be attributed to the complexity of certain vulnerabilities, where the model's attention might be drawn to code elements that are related to the vulnerability but not directly part of the vulnerable lines.

Despite these minor inconsistencies, the high localization accuracy (87%) demonstrates the effectiveness of our attention-based approach in pinpointing vulnerable code sections. This accurate localization provides valuable guidance to the LLM, enabling it to focus on the problematic area and generate more targeted code fixes.

6.3 LLM-Generated Patch Evaluation: A Multi-Perspective Assessment

We evaluated the quality of the patches generated by Google Gemini Pro using three distinct evaluation methods: human evaluation, static analysis with Bandit, and re-evaluation using our trained GAT model. This multi-perspective assessment provides a comprehensive understanding of the effectiveness and reliability of the generated patches.

6.3.1 Evaluation Methodology

We selected 100 code snippets from the test set where the GAT model (trained on CPGs) correctly predicted a vulnerability, and our attention-based localization accurately identified the vulnerable span. For each of these snippets, we generated prompts following the structure described in the previous chapter. These prompts were provided to the Gemini Pro LLM to generate candidate patches.

Each generated patch was then evaluated using the following methods:

1. **Human Evaluation:** A security expert manually reviewed each patch, assessing:
    - **Correctness:** Whether the patch successfully addressed the identified vulnerability.
    - **Functional Equivalence:** Whether the patched code maintained the original functionality of the snippet.
    - **Code Quality:** Whether the patch adhered to good coding practices and did not introduce any new issues, such as syntax errors or logical flaws.
2. **Static Analysis with Bandit:** We re-ran Bandit [15] on the patched code snippets to automatically check for the presence of the original vulnerability

and to identify any new vulnerabilities that might have been introduced by the patch.

3. **Re-Evaluation with GAT Model:** We transformed the patched code into CPG representations and re-evaluated them using our trained GAT model. This assessed whether the patched code was still classified as vulnerable by our model, indicating potential limitations in the LLM's ability to fully address the vulnerability.

6.3.2 Results and Analysis

Table 6.3 summarizes the results of our LLM-generated patch evaluation. The human evaluation revealed that 78% of the patches were deemed correct, effectively addressing the identified vulnerabilities without introducing new issues or altering the original functionality. Bandit analysis confirmed these findings, with 75% of the patched snippets no longer triggering the original vulnerability warnings. Re-evaluation with our trained GAT model further supported these results, with 79% of the patched snippets now classified as non-vulnerable.

Table 6.3: LLM-Generated Patch Evaluation Results

Evaluation Method	**Success Rate**
Human Evaluation (Correctness)	78%
Bandit Analysis (No Original Vulnerability)	75%
GAT Model Re-evaluation (Non-Vulnerable)	79%

These results demonstrate the promising capabilities of LLMs in generating code patches for security vulnerabilities when guided by our framework. The high success rate across multiple evaluation methods suggests that our approach, which combines vulnerability detection with targeted context extraction, provides the LLM with sufficient information to generate relevant and effective fixes.

However, it is important to acknowledge that not all generated patches were successful. The remaining patches (around 20-25%) either failed to fully address the vulnerability, introduced new issues, or altered the original functionality of the code. This highlights the inherent limitations of current LLMs in fully understanding the nuances of security vulnerabilities and the complexities of code repair. Further research is needed to improve the accuracy and reliability of LLM-generated patches, potentially by incorporating more sophisticated reasoning capabilities or by providing even richer contextual information to the LLM.

6.4 Sample Result: End-to-End Vulnerability Detection and Patching

To demonstrate the end-to-end functionality of our framework, we present an illustrative example showcasing the process of vulnerability detection, localization, prompt generation, and LLM-based patch generation.

6.4.1 Input Code Snippet

Consider the following Python code snippet, which contains a Cross-Site Scripting (XSS) vulnerability:

```python
def display_user_profile(username):
    profile_html = f"""
    <h1>Welcome, {username}!</h1>
    <p>This is your profile page.</p>
    """
    return profile_html
```

This snippet represents a common scenario in web applications where user-supplied data is directly incorporated into HTML output without proper sanitization. The vulnerability arises because a malicious user could provide a username containing JavaScript code, which would then be executed in the browser of other users viewing the generated HTML.

6.4.2 GAT Model Prediction and Localization

Our trained GAT model, using the CPG representation of this code, correctly predicts the presence of an XSS vulnerability. The attention-based localization mechanism highlights the following line as the most likely location of the vulnerability:

```
def display_user_profile(username):
    profile_html = f"""
    <span style="color:red">"---"</span>
    <h1>Welcome, {username}!</h1>
    <p>This is your profile page.</p>
    """
    return profile_html
```

This accurate localization is crucial because it directs the LLM's attention to the specific line of code that needs to be modified.

6.4.3 Generated LLM Prompt

Based on the GAT model's prediction and localization, the following prompt is generated for the Google Gemini Pro LLM:

```
### Task: Fix a security vulnerability in the following Python code.

The code has been identified as potentially containing a **Cross-Site
    Scripting (XSS)** vulnerability.
The vulnerable part of the code is highlighted below:
```

~~~~
```python
def display_user_profile(username):
    profile_html = f"""
    <span style="color:red">"---"</span>
    <h1>Welcome, {username}!</h1>
    <p>This is your profile page.</p>
    """
    return profile_html
```
Please provide a corrected version of the entire code that addresses this
    vulnerability while maintaining the original functionality:
```python
def display_user_profile(username):
    profile_html = f"""
    <h1>Welcome, {username}!</h1>
    <p>This is your profile page.</p>
    """
    return profile_html

**Important:**
- Focus on fixing the specific vulnerability identified.
- Ensure the patched code is functionally equivalent to the original code.
- If the vulnerability cannot be fixed without more context, please
    explain why.
~~~~

The prompt clearly instructs the LLM to fix the XSS vulnerability, provides the original code, highlights the vulnerable line, and emphasizes the need to maintain the original functionality.

6.4.4 LLM-Generated Patch

The Google Gemini Pro model, in response to the prompt, generated a comprehensive analysis of the vulnerability and a suggested patch:

LLM Response

Understanding the XSS Vulnerability

The issue: The provided code directly inserts the ‘username‘ variable into the HTML without any sanitization or escaping. This makes it susceptible to Cross-Site Scripting (XSS) attacks. If a malicious user provides a crafted input for the ‘username‘, they could inject malicious script into the generated HTML, which would be executed on the client side when the page is rendered.

Fixing the Vulnerability

To prevent XSS, we need to properly escape the ‘username‘ before inserting it into the HTML. Python’s ‘html‘ module provides the necessary function for this:

```python
import html


def display_user_profile(username):
    profile_html = f"""
    <h1>Welcome, {html.escape(username)}!</h1>
    <p>This is your profile page.</p>
    """
    return profile_html
```

Explanation: - **Import ‘html‘:** This imports the necessary module for HTML escaping. - **‘html.escape(username)‘:** Escapes special characters in the ‘username‘ to prevent XSS. - **f-string:** The f-string is used to format the HTML string with the escaped username.

Additional Considerations

While this code addresses the specific XSS vulnerability in the provided example, it's essential to consider other potential XSS attack vectors and implement broader security measures:

- **Input Validation:** Always validate user input to prevent unexpected or malicious data from being processed. - **Output Encoding:** Consistently encode output to prevent XSS, SQL injection, and other injection attacks. - **Content Security Policy (CSP):** Implement a CSP to further restrict the execution of scripts and other resources. - **Regular Security Audits:** Conduct regular security assessments to identify and address vulnerabilities.

By following these guidelines, you can significantly enhance the security of your web application.

Note: This solution assumes that the 'username' is directly obtained from user input. If the 'username' is retrieved from a trusted source (e.g., a database), the risk of XSS might be mitigated, but it's still recommended to escape the output for best practices.

Would you like to explore additional security measures or discuss a specific use case?

The LLM not only provided a corrected code snippet that uses the html.escape function to sanitize the user input but also included a detailed explanation of the vulnerability, the fix, and additional security considerations. This demonstrates a deeper level of understanding and response from the LLM, making the patch more informative and actionable for developers.

6.5 Discussion

This chapter presented a comprehensive evaluation of our proposed framework, demonstrating its effectiveness in identifying, localizing, and guiding the patching of code vulnerabilities. The key findings from our evaluation are:

- **CPG Representation Superiority:** The GAT model trained on CPGs consistently outperformed the model trained on ASTs, highlighting the importance of leveraging richer semantic information for vulnerability detection.
- **Attention-Based Localization Accuracy:** Our attention-based localization technique accurately identified vulnerable code spans in 87% of the analyzed cases, providing valuable guidance for targeted code fixes.
- **LLM-Generated Patch Effectiveness:** LLM-generated patches, guided by our framework, achieved a high success rate (around 75-80%) in addressing identified vulnerabilities, demonstrating the potential of LLMs for automated code repair.

These results suggest that our hybrid approach, integrating deep learning-based vulnerability detection with LLM-powered code generation, offers a promising avenue for enhancing code security and improving the reliability of software systems. However, it is crucial to acknowledge the limitations of current LLMs in fully understanding the intricacies of code vulnerabilities and repair. Also, a larger and more detailed dataset can further enhance the GAT model as well, improving the overall effectiveness of the proposed system. Further research is needed to enhance the accuracy and reliability of LLM-generated patches and to address the remaining challenges in automating code security.

CHAPTER VII

Conclusion and Future Work

7.1 Concluding Remarks

The pervasiveness of software in all aspects of modern society underscores the critical importance of developing robust and automated solutions for identifying and addressing security vulnerabilities. Our research has made significant strides in this direction by demonstrating the effectiveness of combining graph-based deep learning and LLMs for automated vulnerability detection and rectification. This hybrid approach, leveraging the strengths of both techniques, offers a promising avenue for enhancing code security and improving the reliability of software systems.

This thesis has presented a novel framework for automated vulnerability detection and rectification, addressing a critical need in modern software development. Our approach leverages the strengths of graph neural networks, specifically Graph Attention Networks (GATs), for precise vulnerability detection and the generative capabilities of Large Language Models (LLMs) for automated code repair. Recognizing the limitations of existing techniques, we designed a framework that not only accurately identifies vulnerabilities but also provides targeted guidance to LLMs, ensuring the generation of secure and effective code fixes.

Our research has yielded several key findings that underscore the effectiveness and potential of our proposed framework. Our experiments unequivocally demonstrated that using Code Property Graphs (CPGs) as the code representation significantly enhances vulnerability detection accuracy compared to using Abstract Syntax Trees (ASTs). Furthermore, our attention-based localization technique, leveraging the attention weights learned by our GAT model, exhibited remarkable accuracy in pinpointing the vulnerable code sections within snippets correctly classified as vulnerable. Our evaluation of LLM-generated patches, guided by our framework, revealed another promising finding. We observed a high success rate, ranging from 75% to 80%, in generating correct and effective fixes for the identified vulnerabilities. This result underscores the significant potential of LLMs for automated code repair when provided with accurate contextual information about the vulnerability and its precise location within the code.

Our work makes several significant contributions to the field of automated vulnerability detection and rectification. We provide compelling empirical evidence of the effectiveness of CPGs as a code representation for vulnerability detection. We introduce and validate an attention-based localization technique that effectively pinpoints vulnerable code sections. Most importantly, we demonstrate the viability of a hybrid approach that combines the strengths of graph-based deep learning and LLMs, offering a promising direction for automating code security.

7.2 Limitations and Future Work

While our framework has shown promising results, there are limitations and areas for future research that can further advance the field of automated code security.

One limitation lies in the size and diversity of our dataset. Although comprehensive, further expansion to incorporate code from diverse programming languages, software domains, and vulnerability types would significantly enhance the generalizability and robustness of our model. A larger and more diverse dataset would enable the model to learn from a wider range of coding practices, vulnerability patterns, and semantic contexts, making it more adaptable to real-world codebases.

Another area for improvement is the scalability of CPG construction. Building CPGs for large codebases can be computationally expensive, potentially limiting the applicability of our approach to massive software projects. Investigating more efficient methods for CPG construction or exploring techniques for selectively generating CPGs for specific code sections, particularly those flagged as potentially vulnerable, could enhance the scalability of our framework.

There is also room for improvement in the quality and reliability of LLM-generated patches. While our framework provides targeted guidance to LLMs, achieving consistently accurate and secure code fixes remains a challenge. Incorporating more sophisticated reasoning capabilities into LLMs, such as symbolic execution or formal verification, could lead to more robust and trustworthy code repairs. Furthermore, exploring alternative methods for encoding contextual information or experimenting with different prompt engineering techniques might also enhance the LLM's understanding of the vulnerability and its impact on the code.

The current scope of our framework is limited to addressing individual vulnerabilities. Future research could explore extending it to handle more complex scenarios, such as vulnerabilities that involve the interaction of multiple code components or those requiring a sequence of patches to be fully addressed. Finally, while we conducted human evaluation of the generated patches, a more extensive and systematic human-in-the-loop evaluation would be highly beneficial. Additionally, integrating our framework into real-world development workflows and conducting user studies to assess its usability and effectiveness in practice would provide valuable insights into its real-world impact and guide further improvements.

By addressing the identified limitations and pursuing the proposed directions for future work, we can strive towards more comprehensive and reliable automated code security solutions, contributing to the development of safer and more resilient software systems that are essential for the functioning of our increasingly digital world.

Bibliography

[1] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2003.

[2] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A.-R. Sadeghi, A. Bartel, J. Klein, and Y. Le Traon, "An empirical comparison of static bug finders for android," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 164–175, 2016.

[3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 3–12, 2010.

[4] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Indianapolis, IN: Addison-Wesley Professional, 2007.

[5] P. Oehlert, "Making fuzzing a first-class citizen in security testing," in *Proceedings of the 2011 IEEE/ACM International Conference on Automated Software Engineering*, pp. 483–486, 2011.

[6] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Pearson Education Limited, 2015.

[7] J. Li, Y. Zhou, S. Xu, H. Liu, Y. Fang, and X. Guan, "Vuldeepecker: A deep learning-based system for vulnerability detection," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.

[8] A. Caliskan, A. Narayanan, and J. Harer, "Cve-cwe mapping using natural language processing and graph embedding techniques," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 477–487, IEEE, 2021.

[9] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, *et al.*, "Codebert: A pre-trained model for programming language representation learning," *arXiv preprint arXiv:2002.08155*, 2020.

[10] Y. Zhou, S. Liu, X. Du, X. Xie, and H. Peng, "Devign: Effective vulnerability identification by learning node representations for cpgs," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1413–1428, 2019.

[11] Y. Li, S. Zheng, Y. Pei, and S. He, "Graph representation learning for code analysis tasks: A survey," *ACM Computing Surveys (CSUR)*, vol. 56, no. 4, pp. 1–56, 2023.

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

[13] D. Yan, A. Rountev, and S. Malik, "Code property graphs: Towards a unified system for program analysis," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 931–942, ACM, 2016.

[14] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[15] PyCQA, "Bandit - Security linter for Python code." `https://github.com/PyCQA/bandit`. Accessed: [Date you last accessed the Bandit GitHub repository].

[16] MITRE Corporation, "Common weakness enumeration (cwe)."

[17] OWASP, "Cross-site scripting (xss)." `https://owasp.org/www-project-top-ten/2017/A7_Cross-Site_Scripting_(XSS).html`. Accessed: Oct. 27, 2023.

[18] OWASP, "Sql injection." `https://owasp.org/www-project-top-ten/2017/A1_Injection.html`. Accessed: Oct. 27, 2023.

[19] OWASP, "Session fixation." `https://owasp.org/www-project-top-ten/2013/A2-Broken_Authentication_and_Session_Management.html#Session_Fixation`. Accessed: Oct. 27, 2023.

[20] OWASP, "Path traversal." `https://owasp.org/www-community/attacks/Path_Traversal`. Accessed: Oct. 27, 2023.

[21] OWASP, "Os command injection." `https://owasp.org/www-community/attacks/Command_Injection`. Accessed: Oct. 27, 2023.

[22] J. Zhou, G. Cui, Z. Zhang, C. Y. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[23] C. Sadowski, K. T. Stolee, and S. Elbaum, "Lessons from building static analysis tools at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pp. 1–10, 2018.

[24] A. Shostack, *Threat modeling: designing for security.* John Wiley & Sons, 2014.

[25] B. Chess and J. West, *Secure Programming with Static Analysis.* Addison-Wesley Professional, 2007.

[26] S. K. C. S. P. W. V. Johnson, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, pp. 1–36, 2013.

[27] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium*, vol. 8, pp. 151–166, 2008.

[28] Y. Chen, Z. Cui, X. Wang, J. Xue, and X. Cai, "A survey of fuzzing techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–35, 2018.

[29] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, "Xlm: Generalized masked language models for cross-lingual understanding," in *Advances in neural information processing systems*, vol. 33, pp. 19999–20010, 2020.

[30] S. M. Mousavi, G. Scanniello, F. Paradisi, C. Gaüzère, and M. Thomas, "Static analysis of software vulnerabilities using machine learning," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 489–508, 2015.

[31] X. Pan, H. Chen, H. Zhang, and J. Wu, "Deeppoly: Automatic inference of verifiable functional properties for deep neural networks," *arXiv preprint arXiv:1709.07660*, 2017.

[32] Q. Feng, Y. Zhou, C. Xu, Y. Yu, X. Lu, and B. Xu, "Automated vulnerability detection for web applications based on deep learning," in *Proceedings of the 27th International Conference on World Wide Web*, pp. 845–854, 2018.

[33] N. Al-Ajlan and M. A. Al-Qabban, "Software vulnerability detection using deep neural networks: A survey," *arXiv preprint arXiv:1801.01678*, 2018.

[34] W. Samek, T. Wiegand, and K.-R. Müller, "Explainable ai: interpreting, explaining and visualizing deep learning," *arXiv preprint arXiv:1708.08296*, 2017.

[35] L. Tang, Z. Xu, Z. Yan, Y. Zhou, and Y. Liu, "Cpgsec: A cpg-based approach for identifying security-related code smells," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 301–312, 2020.

[36] Y. Sun, Z. Liu, W. Li, and Y. Sun, "Cpg-vul: Cpg-based vulnerability detection for smart contracts," in *Proceedings of the 2022 International Conference on Software Engineering (ICSE)*, pp. 1541–1552, 2022.

[37] T. Nguyen, A. Nguyen, T. Tran, H. A. Rajan, and T. N. Nguyen, "Graph neural networks for software vulnerability detection: A survey," *ACM Computing Surveys (CSUR)*, vol. 55, no. 8, pp. 1–35, 2022.

[38] R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnnexplainer: Generating explanations for graph neural networks," in *Advances in neural information processing systems*, vol. 32, 2019.

[39] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. E. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03347*, 2021.

[40] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[41] F. Henglein, M. Schafer, M. Menzel, M. Smith, and M. Hirschfeld, "Secure coding practices for large language models: Recommendations from an empirical study," in *Proceedings of the 2021 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1498–1511, 2021.

[42] J. Pearl and D. Mackenzie, "The seven deadly sins of ai predictions," *ACM Computing Surveys (CSUR)*, vol. 55, no. 4, pp. 1–36, 2022.

[43] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proceedings of the 2013 conference on empirical methods in natural language processing*, pp. 1631–1642, 2013.

[44] M. Tufano, C. Watson, G. Bavota, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 552–562, 2019.

[45] E. Dinella, C. Henning, J. Si, and Z. Su, "Learning to rank code-change suggestions for automated program repair," *arXiv preprint arXiv:2004.05827*, 2020.

[46] T. Lutellier, M. Tan, Y. Qi, S.-W. Zhou, and D. Poshyvanyk, "Coconut: Combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1021–1033, 2020.

[47] SonarSource, "Sonarqube." https://www.sonarqube.org/. Accessed: [Date you last accessed the website].

[48] r2c, "Semgrep: A fast, open-source, static analysis tool for finding and preventing security vulnerabilities." https://semgrep.dev/.

[49] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[50] F. E. Allen, "Control flow analysis," *SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[51] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[52] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[53] "SARD - Software Assurance Reference Dataset." https://samate.nist.gov/SARD/. Accessed: [Date you last accessed the SARD website].

[54] W. G. Halfond, J. Viegas, and A. Orso, "Sql injection attacks and defense techniques," in *Proceedings of the 2006 international workshop on dynamic analysis*, pp. 1–7, 2006.

[55] M. Zalewski, *Cross site scripting attacks: Cross-site scripting vulnerabilities and techniques to prevent them*. No Starch Press, 2002.

[56] OWASP, "Command injection," 2023. https://owasp.org/www-project-top-ten/2021/A01-Broken_Access_Control.html.

[57] OWASP, "Path traversal," 2023. https://owasp.org/www-community/attacks/Path_Traversal.

[58] P. S. Foundation, "ast — abstract syntax trees." https://docs.python.org/3/library/ast.html.

[59] S. Security, "Joern: Open-source code analysis platform for c/c++/java/binary/javascript based on code property graphs." https://joern.io/.

[60] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[61] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[62] G. AI, "Gemini pro: The next generation of ai models." https://ai.google.dev/tutorials/llm/gemini.