People's Democratic Republic of Algeria

Ministry of Higher Education and Scientific Research

University of 8 Mai 45 Guelma

Faculty of Mathematics, Computer Science and Material Science

Computer Science Department

MASTER'S THESIS:

Branch: Computer Science

Option: Informatic System

Topic:

Leveraging Large Language Models For Automated Software Vulnerability Detection and Analysis

Presented By:

BOUCENA Amina

In Front of The jury:

Dr. BENAMIRA Adel	President
Pr. BENCHERIET Chemesse Ennahar	Examiner
Dr. FERRAG Mohamed Amine	Supervisor

June 2025

Abstract:

The growing integration of software across various sectors has amplified the urgency to develop effective mechanisms for safeguarding systems against sophisticated cyberattacks. This thesis investigates the application of modern artificial intelligence specifically transformer-based models for improving software vulnerability detection.

The research focuses on designing intelligent algorithms capable of autonomously analyzing source code and identifying subtle security flaws. It emphasizes automated detection and fine-grained classification of vulnerabilities. Our methodology includes constructing high-quality labeled datasets and training advanced models such as BERT and RoBERTa to extract threat patterns and detect weaknesses at the line level and across multiple vulnerability categories.

Experimental results show that transformer-based models significantly outperform traditional methods in both detection accuracy and analysis efficiency. Moreover, the practical deployment of our model in real-world scenarios confirms its utility in supporting software security analysts. This work thus provides a meaningful contribution to the field of cybersecurity, offering a scalable and robust solution to the persistent challenges in securing modern software systems.

Keywords: Large Language Models (LLMs), Software Vulnerabilities, Cybersecurity, Deep Learning, Code Analysis, Vulnerability Detection, Secure Software Development

الملخص

ملخص المذكرة أدى التوسع المتزايد في استخدام البرمجيات ضمن مختلف القطاعات إلى بروز الحاجة الماسة إلى تطوير آليات فعّالة لحماية هذه الأنظمة من الهجمات السيبرانية المتطورة. في هذا السياق، تتناول هذه المذكرة، توظيف تقنيات الذكاء الاصطناعي الحديثة، وخاصة نماذج المحولات ، لتحسين قدرات أنظمة اكتشاف الثغرات الأمنية.

يرتكز العمل على تصميم خوارزميات ذكية قادرة على تحليل التعليمات البرمجية تلقائيًا والتعرف على الثغرات الدقيقة، مع التركيز على الكشف التلقائي والتصنيف الدقيق للثغرات. تم بناء منهجية البحث على إعداد مجموعات بيانات عالية الجودة وغيرها، لاستخلاص الأنماط الخطرة وتحديد RoBERTaو BERT وتمثيلها بشكل مناسب، ثم تدريب نماذج متقدمة مثل نقاط الضعف البرمجية على مستوى السطر وضمن تصنيفات متعددة. تشير نتائج التجارب إلى أن النماذج المعتمدة على المحولات تحقق أداءً متفوقًا مقارنة بالأساليب التقليدية، سواء من حيث دقة الكشف أو سرعة التحليل. كما يثبت هذا العمل فعالية هذه النماذج عند تطبيقها في بيئات واقعية، مما يمهد الطريق لاعتمادها مستقبلاً في أنظمة الحماية البرمجية. بذلك، تقدم هذه الدراسة مساهمة نوعية في مجال الأمن السيبراني، من خلال تقديم نموذج متطور وعملي لمواجهة تحديات تأمين الثغرات البرمجية، الأمن السيبراني، التعلم ،(LLMs) البرمجيات الحديثة. الكلمات المفتاحية: النماذج اللغوية الكبيرة العميق، تحليل الشيفرة البرمجية، كشف الثغرات، تطوير البرمجيات الآمن.

الكلمات المفتاحية:النماذج اللغوية الكبيرة, الثغرات البرمجية , الامن السيبراني, التعلم العميق, تحليل الشيفرة البرمجية, كشف الثغرات , تطوير البرمجيات الامن.

Résumé

L'intégration croissante des logiciels dans divers secteurs a rendu essentielle la mise en place de mécanismes efficaces pour protéger les systèmes contre les cyberattaques sophistiquées. Ce mémoire explore l'application de l'intelligence artificielle moderne notamment les modèles basés sur les transformeurs pour améliorer la détection des vulnérabilités logicielles.

Le travail vise à concevoir des algorithmes intelligents capables d'analyser automatiquement le code source et d'identifier les failles de sécurité les plus subtiles. Il met l'accent sur la détection automatisée et la classification fine des vulnérabilités. La méthodologie adoptée repose sur la création de jeux de données étiquetés de haute qualité, ainsi que sur l'entraînement de modèles avancés tels que BERT et RoBERTa pour détecter les schémas de menaces au niveau de la ligne de code et à travers plusieurs catégories de vulnérabilités.

Les résultats expérimentaux démontrent que les modèles basés sur les transformeurs surpassent largement les méthodes traditionnelles, tant en précision qu'en efficacité d'analyse. De plus, le déploiement du modèle dans un contexte réel confirme sa pertinence pour les analystes en sécurité logicielle. Ce travail constitue ainsi une contribution notable au domaine de la cybersécurité, en proposant une solution robuste et évolutive face aux défis récurrents de la protection des systèmes logiciels modernes.

Mots-clés: Modèles de langage de grande taille (LLMs), vulnérabilités logicielles, cybersécurité, apprentissage profond, analyse de code, détection de vulnérabilités, développement logiciel sécurisé

.

.

Acknowledgement:

I thank Allah, the Most Merciful and Almighty, who guided and helped me complete this work. I pray that this effort will be accepted and rewarded. May peace and blessings be upon our beloved Prophet Muhammad, the best of all creation, and upon his family and companions.

I would like to extend my heartfelt thanks to my supervisor, Mr. Ferrag, for his invaluable guidance, continuous support, and insightful advice throughout every stage of this thesis. His encouragement and expertise greatly contributed to the quality and direction of this work. I am truly grateful for the opportunity to learn under his supervision.

I am also deeply thankful to the members of the jury for accepting to evaluate this work, and for their valuable time, observations, and constructive remarks.

My appreciation further goes to all the professors of the Computer Science Department who enriched my academic journey with knowledge and inspiration. Their dedication to teaching has left a lasting impact on my formation.

Dedication:

To the light of my life ...

To my beloved parents, who stood by me through every phase of my life.

Your unwavering emotional, and spiritual support has been the greatest blessing I've ever known. Without your encouragement and prayers, none of this would have been possible. This work is as much yours as it is mine.

To my brothers, whose presence was always a source of joy and strength. Your smiles, laughter, and pride lifted me up and kept me going.

To my dear friends, and in particular Nor and Chaima, who have become an inseparable part of my academic journey. From strangers to sisters, our friendship blossomed during these university years and added warmth and meaning to every step.

This work is dedicated to you all, with love and deep gratitude.

Content

General introduction 1
1. Chapter 01 3
1.1. Introduction: 3
1.2. Common Software Security Vulnerabilities: 3
1.3. Vulnerability Detection Techniques: 11
1.3.1. Static analysis: 11
1.3.2. Dynamic Analysis: 12
1.3.3. Formal Verification: 12
1.3.4. AI and Machine Learning(ML): 12
1.4. Conclusion: 13
2. Chaptre 02 14
2.1. Introduction 14
2.2. Historical Timeline of Vulnerability Detection: 14
2.3. Machine Learning-Based Approaches: 15
2.4. RNN-Based Vulnearbility Detection: 16
2.5. Deep Learning Models: 19
2.6. Transformer-Based Models: 19
2.6.1. Code-oriented Transformers: 20
2.6.2. Fine-Grained Vulnerability Localization: 20
2.6.3. Lightweight and Specialized Transformers: 21
2.6.4. Decoder-Only Models: 21
2.7. Datasets and Benchmarks: 22
2.8. Benchmarks and Evaluation Metrics: 24
2.9. Use In Industry: 25
2.10. Related Work: 26
2.11. Conclusion : 27
3. Chaptre 03 28
3.1. Introduction: 28
3.2. Proposed LLM-Based Vulnerability Detector: 28
3.2.1. Overall System Architecture: 28
3.2.2. Data Acquisition and Labeling: 29
3.2.2.1. Dataset Source: FormAI-v2 30
3.2.2.2. Labeling Methodology: 30
3.2.2.3. Dataset Characteristics: 30
3.2.3. Preprocessing and Input Representation: 31
3.2.4. LLM Fine-Tuning Strategy: 31
3.2.5. Inference Pipeline: 32
3.3. Experimental Setup and Methodology 33
3.3.1. Dataset Preparation: 33
3.3.2. Fine-Tuning Environment: 33
3.3.3. Evaluation Metrics: 34
3.4. Experimental Results: 35
3.4.1. Training & Validation Curves: 35
3.4.2. Test-Set Performance: 37
3.4.3. Confusion Matrix & Error Analysis: 40
3.4.4. Comparison with Baselines: 45
3.5. Discussion 48
3.5.1. Impact of Prompt Design and Context Length: 48
3.5.1.1. Prompting Strategies in Vulnerability Detection: 48
3.5.2. Strengths and Limitations of LLM-Based Detection: 49
3.5.2.1. Strenghts: 49
3.5.2.2. Limitation: 49
3.5.3. Scalability, Inference Latency & Resource Consumption: 50
3.6. Chapter Summary and Future Directions: 51

3.6.1. Chapter Summary: 51
3.6.2. Future Directions: 51
General Conclusion 53
Bibliography: 54

List of Tables

Table 1 :Recent Studies on Vulnerability Detection Using LLMs 26
Table 2 :BERT Training and Validation Results 35
Table 3 :RoBERTa Training and Validation Results 36
Table 4 :DeBERTa Training and Validation Results 37
Table 5 :NeoBERT Training and Validation Results 37
Table 6 :Classification Report for BERT 38
Table 7 :Classification Report for CodeBERT 38
Table 8 :Classification Report for RoBERTa 39
Table 9 :Classification Report for DeBERTa 39
Table 10 :Classification Report for NeoBERT 40
Table 11 :Test-Set Evaluation Results for Random Forest 45
Table 12 :Test-Set Evaluation Results for Logistic Regression 46
Table 13 :Test-Set Evaluation Results for XGBoost 46
Table 14 :Test-Set Evaluation Results for Naive Bayes 47
Table 15 :Test-Set Evaluation Results for SVC 47
Table 16 :Comparison of the Proposed Model with LLM Models and Traditional ML 48
Table 17 :Comparison of Prompt Styles and Context Lengths on BERT-Based Vulnerability Classification 49

List of Figures

Figure 1 :Illustration of an Out-of-Bounds Write Vulnerability and Its Potential Consequences 4

Figure 2 :Illustration of a Type Confusion Vulnerability and Its Potential Consequences 5

Figure 3 : Illustration of an OS Command Injection Vulnerability and Its Consequences 6

Figure 4 :Illustration of a Code Injection Vulnerability via eval() and Its Consequences 6

Figure 5 : Illustration of an Insecure Deserialization Vulnerability and Its Potential Impact 7

Figure 6 :Illustration of a Path Traversal Vulnerability Leading to Unauthorized File Access 8

Figure 7 :Illustration of a Missing Authentication for Critical Function Vulnerability 9

Figure 8 :Illustration of an SQL Injection Vulnerability Exploiting Unsanitized User Input 10

Figure 9 :Illustration of a Use-After-Free Vulnerability Leading to Memory Corruption 10

Figure 10 :Illustration of Command Injection via Unsanitized Input in Dynamically Constructed Shell Commands 11

Figure 11 :Timeline of key research milestones in vulnerability detection using static analysis, machine learning, and formal methods (2004–2025). 15

Figure 12 :Taxonomy of Machine Learning and Transformer-Based Models for Vulnerability Detection 16

Figure 13 :Internal Architecture of the BLSTM-Based Classifier in VulDeePecker[25] 17

Figure 14 :Overview of the SySeVR Framework for Deep Learning-Based Vulnerability Detection[33] 17

Figure 15 :Architecture of μVulDeePecker Training and Inference Phases[34] 18

Figure 16 :Overall System Architecture for Prompt-Based Vulnerability Classification using a Transformer Model 28

Figure 17 :Confusion Matrix of BERT on the Top 10 Most Frequent CWE Vulnerability Classes . 41

Figure 18 :Confusion Matrix of CodeBERT on the Top 10 Most Frequent CWE Vulnerability Classes 42

Figure 19 :Confusion Matrix of RoBERTa on the Top 10 Most Frequent CWE Vulnerability Classes 43

Figure 20 :Confusion Matrix of DeBERTa on the Top 10 Most Frequent CWE Vulnerability Classes 44

Figure 21 :Confusion Matrix of NeoBERT on the Top 10 Most Frequent CWE Vulnerability Classes 45

General introduction

In the modern digital era, software has become the backbone of nearly every sector, from finance and healthcare to transportation, education, and communication. Whether in smartphones, embedded systems, or complex cloud architectures, software-driven systems are essential for running and maintaining daily operations. However, the rapid expansion of software usage has also made it a primary target for malicious attacks. Vulnerabilities in software code, especially when left undetected, present serious security risks that can lead to data breaches, financial losses, and compromise of critical infrastructures.

Traditionally, the detection of software vulnerabilities has relied heavily on static analysis, dynamic testing, and manual code reviews. While these methods offer some level of protection, they are often limited in scope and struggle to keep up with the increasing complexity and size of modern codebases. Furthermore, static tools may generate false positives, while dynamic methods often miss corner cases or require extensive execution environments. As such, the software security community has begun to explore more intelligent and scalable solutions.

Recent advancements in Artificial Intelligence (AI) and Natural Language Processing (NLP) especially with the advent of Large Language Models (LLMs)have opened up new possibilities for analyzing code as if it were a form of structured language. Models like BERT, RoBERTa, and specialized variants like CodeBERT and NeoBERT have demonstrated significant potential in understanding code semantics, identifying patterns associated with vulnerabilities, and classifying code fragments with high accuracy. These models, originally designed for human language understanding, can be fine-tuned on code corpora to perform tasks such as vulnerability detection, code summarization, and bug localization.

This thesis aims to leverage the capabilities of LLMs to build an effective software vulnerability detection system. The core objective is to classify C source code as either secure or vulnerable, and, in the case of vulnerability, identify the specific type of vulnerability using a fine-tuned transformer-based model. To achieve this, we compare different LLMs and prompting strategies, investigate the impact of input context length, and assess the models' generalization capabilities.

The work is organized into three main chapters:

Chapter 1: lays the foundational concepts for this study. It begins with a formal definition of software vulnerabilities and explores the most frequently occurring types, based on established taxonomies such as CWE (Common Weakness Enumeration) and KEV (Known Exploited Vulnerabilities). This chapter also discusses traditional techniques for vulnerability detection, such as static analysis, dynamic analysis, and symbolic execution, providing the necessary background to understand the motivation for AI-based alternatives.

Chapter 2: provides a comprehensive review of the literature and technological background relevant to our work. It traces the evolution of machine learning models applied to software analysis from early neural architectures like RNNs and CNNs, to modern transformer-based models. We discuss models pre-trained on code, such as CodeBERT, GraphCodeBERT, and RoBERTa, and evaluate how they are adapted to the domain of code intelligence. In addition, we present a survey of widely-used datasets in vulnerability detection research, including Juliet, Draper, Devign, VulDeePecker, and most notably, FormAI, which serves as the backbone for our experimental setup. Finally, the chapter highlights previous research efforts in the field, comparing methodologies and objectives to our own approach.

Chapter 3: introduces our proposed LLM-based detection system and provides a detailed breakdown of the methodology. We start by preprocessing and filtering the FormAI dataset, selecting the most frequent error types and cleaning the labels. We then experiment with two different prompting strategies Prompt A, where the model is asked a direct question (e.g., “What type of vulnerability is this code?”), and Prompt B, a masked-language approach that lets the model predict the most likely completion of a partially filled sentence. The chapter also examines the effect of input context length, comparing the standard 512-token input to a truncated 128-token setup. We conduct extensive experiments on multiple models including BERT, NeoBERT, CodeBERT, and RoBERTa, and evaluate their performance using metrics such as accuracy, precision, recall, F1-score, and confusion matrices. Additionally, the chapter includes in-depth error analysis and visualizations to better understand model behavior.

The thesis culminates in a general discussion where we assess the impact of prompt design, model architecture, and input length on performance. We highlight the strengths and limitations of using LLMs for vulnerability detection, such as their ability to generalize across classes and their sensitivity to context size. We also address resource-related considerations such as training time and memory usage, and discuss practical deployment challenges like scalability, interpretability, and system integration.

Finally, we summarize the key findings of our work and propose several future directions for extending this research, including automatic prompt engineering, integrating static analysis features, and adapting the system to real-world deployment scenarios.

Chapter 01

Cybersecurity and Software Vulnerabilies

1.1. Introduction:

Today, digital systems aren't just tools we use they've become a part of everything. From the phone in your pocket to national infrastructure, software is now deeply embedded in the way we live and work. And with this growing dependency comes a bigger responsibility: cybersecurity is no longer optional it's essential. As systems get more connected and more complex, they also become more exposed, and defending them becomes more challenging than ever[1][2].

What we're seeing in recent research is a shift in mindset: security today isn't just about reacting to threats it's about staying ahead of them. It means spotting vulnerabilities early, blocking attacks before they happen, and responding intelligently when things go wrong. This proactive approach is more important than ever in a world where systems are increasingly fast, distributed, and interdependent[3].

And the risks are real. We've already seen how a simple flaw can lead to major damage. Take Slammer or Blaster for example two worms that infected over 200,000 systems in just a few hours by exploiting basic bugs [3]. And that was just the beginning. Since then, threats have become faster, smarter, and much more sophisticated. The Big-Vul dataset alone contains over 38,000 real vulnerabilities found in C/C++ programs, each linked to known CVE identifiers a clear sign that these flaws are not only common, but also dangerously persistent[4].

What's even more worrying is the speed of modern attacks. Zero-day exploits which take advantage of unknown flaws before anyone has time to patch them highlight just how urgent it is to move from reactive defense to smarter, automated protection. The truth is: traditional tools struggle. They rely on hard-coded rules, often overwhelm developers with false positives, and don't always keep up with the speed of modern software development[5].

1.2. Common Software Security Vulnerabilities:

A software vulnerability can be defined as a flaw or weakness in a computer program that may be exploited to compromise the confidentiality, integrity, or availability of a system. Such vulnerabilities often arise due to programming errors, lack of input validation, or misconfigurations. While some of these flaws may seem minor such as neglecting to validate user input they can be leveraged in combination with other weaknesses to facilitate serious attacks. These issues may result from rushed development, insecure frameworks, or simple human oversight. Understanding the nature and origins of software vulnerabilities is therefore essential in developing secure and resilient code. In what follows, we will present a detailed overview of the Top 10 most critical software vulnerabilities to provide context for the experimental work that follows.

1.2.1. Out-of-bounds Write:

An Out-of-Bounds Write (OOB write) occurs when a program writes data beyond the limits of an allocated buffer, leading to memory corruption. In kernel contexts, it is formally modeled as a triplet (offset, length, value), capturing how far the write goes, how many bytes are written, and what is written[6].

Figure 2 visually demonstrates how an Out-of-Bounds Write vulnerability occurs when a program writes data outside the allocated bounds of a memory buffer. In this example, user input ({AAAA}) is written into a buffer without appropriate boundary checks. The program expects the buffer to handle a fixed amount of data (e.g., buffer[10]), but the input exceeds this limit, resulting in an overflow. The excess data corrupts adjacent memory locations, potentially leading to arbitrary code execution or system crashes. This kind of vulnerability is especially dangerous because the overwritten memory may contain control data such as return addresses or function pointers, making exploitation both feasible and impactful if no memory validation mechanisms are in place.

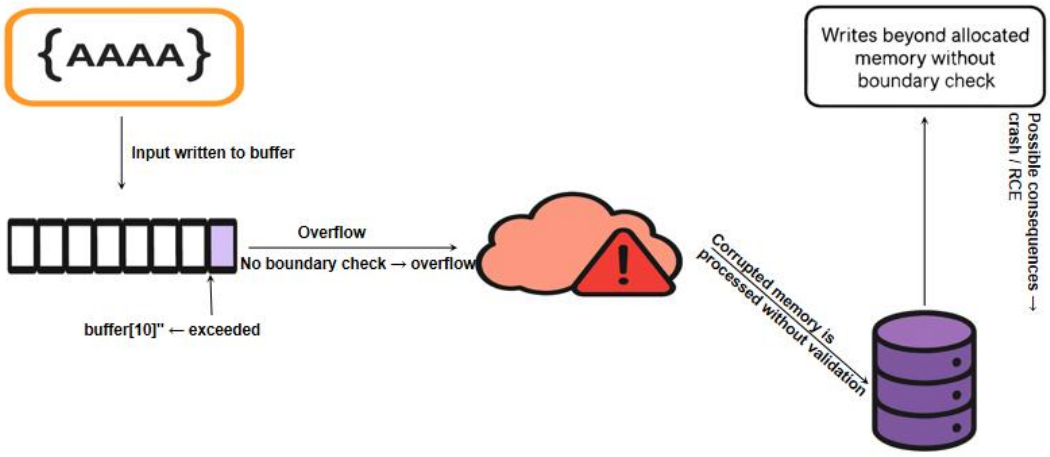

Figure 1:Illustration of an Out-of-Bounds Write Vulnerability and Its Potential Consequences

1.2.2. Type Confusion:

Type confusion occurs when a program uses a resource (such as an object or memory region) as though it were of a different type than it actually is. This mismatch can allow an attacketo execute arbitrary code by manipulating object layouts or invoking unintended functions[7].

Figure 2 illustrates the mechanism of a Type Confusion vulnerability, where an attacker sends a crafted object with a falsified type annotation (e.g., type: B) to a system expecting a different, safe type (e.g., Type A). The system receives this object and, due to the absence of strict type validation, processes it as if it were of the expected type. This leads to a mismatch between the actual and assumed object structure. Consequently, unsafe operations may be performed on the misinterpreted object. The vulnerability stems from treating incompatible memory representations as equivalent, often resulting in logic errors, memory corruption, or even arbitrary code execution particularly if the misused object contains executable data or references.

This figure highlights how type mismatches can lead to critical security breaches in systems that process untrusted input without rigorous type checks.

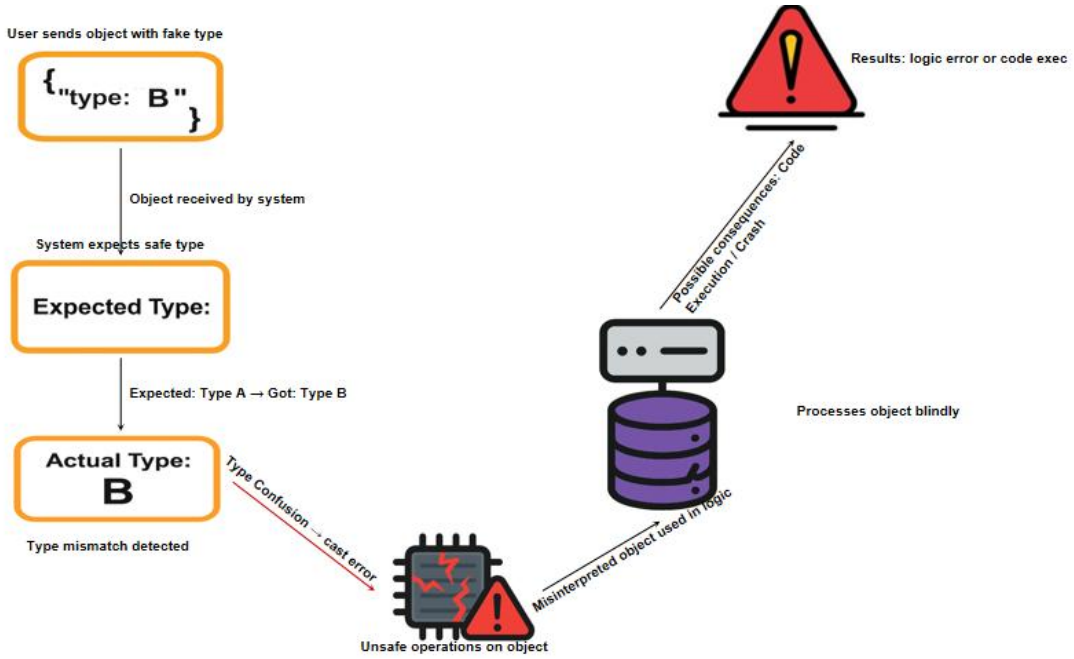

Figure 2:Illustration of a Type Confusion Vulnerability and Its Potential Consequences

1.2.3. OS Command Injection:

OS command injection vulnerabilities arise when input data is not properly validated or sanitized and is passed directly into a command shell. This allows attackers to execute arbitrary commands on the host operating system[8].

Figure 3 depicts the exploitation process of an OS Command Injection vulnerability, where user-supplied input is unsafely incorporated into a system-level command. In this example, the application dynamically constructs a shell command using input like "127.0.0.1" and passes it to a system function such as system(command). Due to the absence of input validation or sanitization, an attacker can manipulate the input to inject arbitrary shell commands. These commands are interpreted and executed by the operating system shell, often with elevated privileges if the application itself has such rights. As shown in the figure, this can lead to arbitrary command execution, allowing the attacker to access, modify, or delete system data, compromise the server, or escalate privileges. This vulnerability is especially critical in web applications where system calls are dynamically built from HTTP parameters or form inputs.

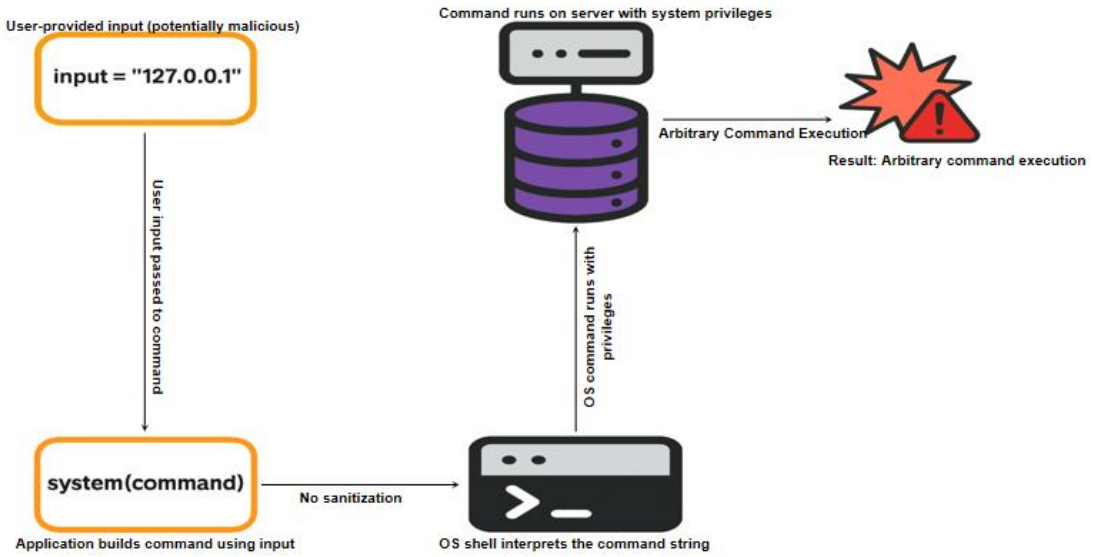

Figure 3: Illustration of an OS Command Injection Vulnerability and Its Consequences

1.2.4. Code Injection:

Code injection vulnerabilities occur when untrusted input is used in code generation or evaluation functions. Attackers exploit these flaws to inject and execute arbitrary code, often through dynamically interpreted languages[8].

Figure 4 demonstrates a Code Injection vulnerability, specifically through unsafe use of the eval() function. In this scenario, untrusted user input such as a Python string containing executable code is directly passed into the eval() function without any validation or sanitization. As a result, the application dynamically evaluates the input as actual code. If the application runs with elevated privileges, this injected code is executed in the system's runtime environment, potentially leading to arbitrary code execution or complete system compromise. This figure highlights the critical danger of using eval() (or similar functions like exec() or Function() in JavaScript) with user-controllable input, as it allows attackers to hijack application behavior and inject malicious instructions into the execution flow.

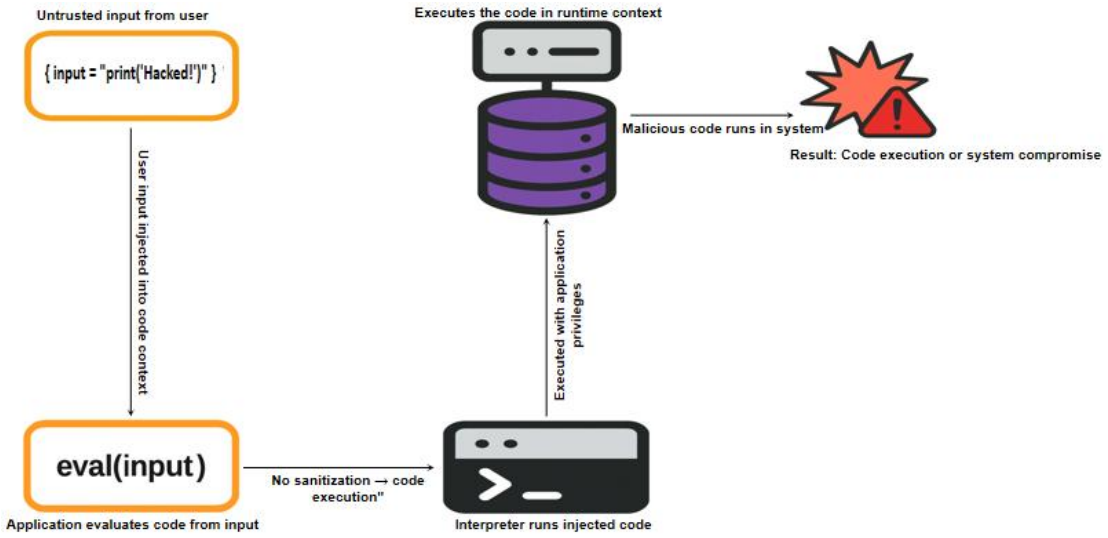

Figure 4:Illustration of a Code Injection Vulnerability via eval() and Its Consequences

1.2.5. Insecure Deserialization:

Insecure deserialization is a critical vulnerability that arises when applications process serialized data from untrusted sources without performing proper validation. By blindly trusting and deserializing such input, the application becomes vulnerable to a range of attacks including arbitrary code execution, data manipulation, or privilege escalation. Attackers can craft malicious objects containing embedded payloads that exploit weaknesses in the deserialization logic[9].

Figure 5 illustrates how an Insecure Deserialization vulnerability can be exploited when a server deserializes untrusted input without validating its type or content. In this scenario, an attacker crafts a malicious serialized object (payload = maliciousObject) and sends it to a vulnerable application. The server-side logic then deserializes this tampered data blindly assuming it to be safe thereby inadvertently instantiating attacker-controlled objects. This can lead to arbitrary code execution, data tampering, or even privilege escalation, depending on the privileges of the application process and the behavior of the deserialized object. The diagram emphasizes the danger of using unsafe deserialization routines in systems that trust user input, especially in languages like Java, Python, or PHP, where deserialization mechanisms can invoke methods during object reconstruction.

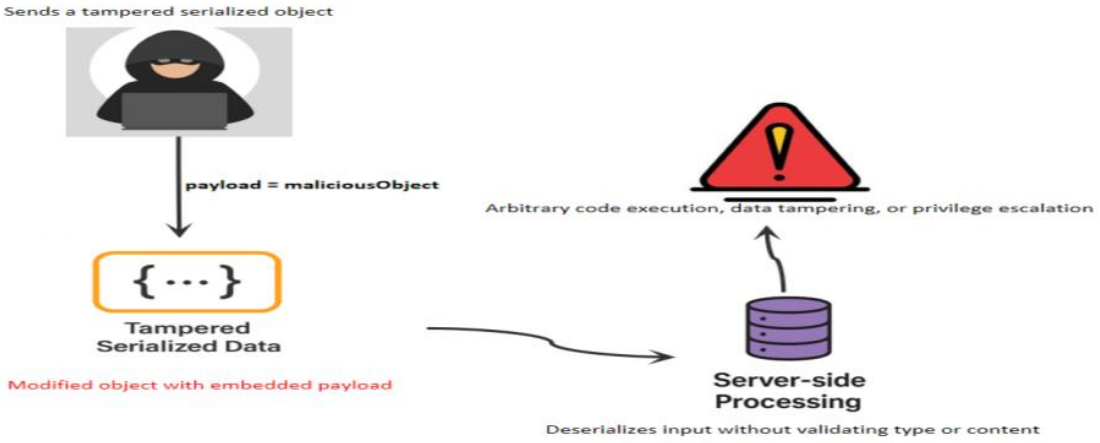

Figure 5: Illustration of an Insecure Deserialization Vulnerability and Its Potential Impact

1.2.6. Directory Traversal:

Directory traversal is a critical security vulnerability that occurs when an application improperly handles user-supplied input used to construct file paths. By injecting directory traversal sequences such as ../, attackers can manipulate file path references to access files and directories outside the intended scope of the application. This vulnerability may allow unauthorized access to sensitive files, including system configurations and user credentials[10].

Figure 6 demonstrates a Path Traversal vulnerability, where a web application improperly trusts user input to construct file paths. In the illustrated scenario, the attacker manipulates the input parameter (e.g., file=../../../../etc/passwd) to traverse directories outside the intended scope. Since the application reads the file path directly from user input without validating or sanitizing it, the attacker can exploit this

to access sensitive files on the server's filesystem. The vulnerability arises when the application concatenates user input with a base directory path and passes it to file handling functions like open(). If no checks are in place to restrict access to authorized directories, the result may be leakage of critical files (e.g., password files, configuration files), leading to unauthorized disclosure, system compromise, or further privilege escalation.

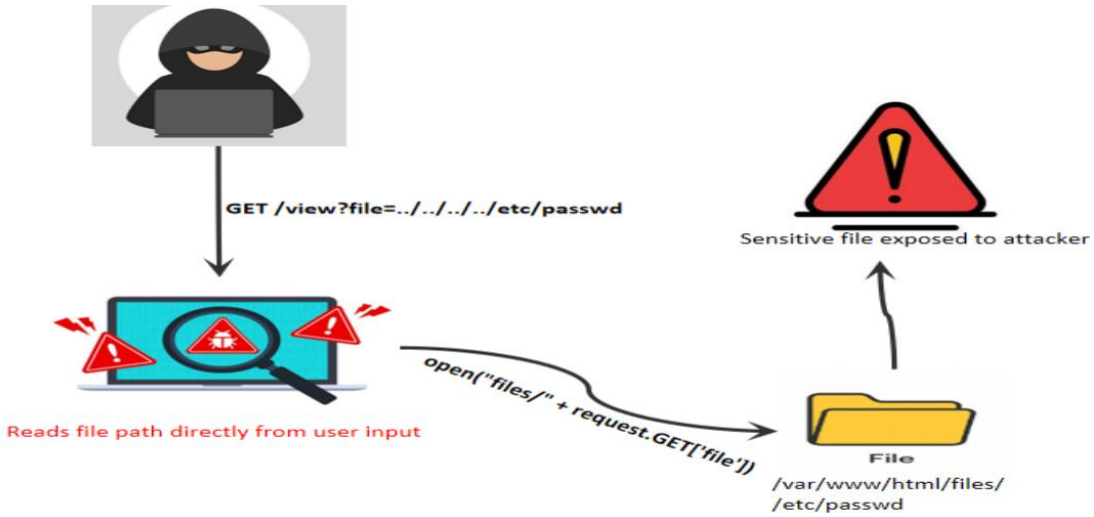

Figure 6:Illustration of a Path Traversal Vulnerability Leading to Unauthorized File Access

1.2.7. Missing Authentication for Critical Function:

A Missing Authentication for Critical Function vulnerability occurs when a system fails to require user authentication before granting access to functionalities that should be restricted. This enables unauthenticated users to perform sensitive operations, possibly leading to unauthorized actions or data exposure[11][12].

Figure 7 demonstrates a Missing Authentication for Critical Function vulnerability, where a sensitive operation such as deleting a user account is exposed to unauthenticated users. In this example, the application provides an endpoint (e.g., /delete-user?id=42) that performs a destructive action but does not enforce authentication or authorization checks. An attacker can exploit this by crafting and sending requests directly to the endpoint without needing to log in or prove their identity. As shown in the figure, the server processes the unauthenticated request and executes the operation, leading to unauthorized access and potential system misuse. This vulnerability typically arises from poor access control design and can have severe consequences, including privilege escalation, data loss, and service disruption.

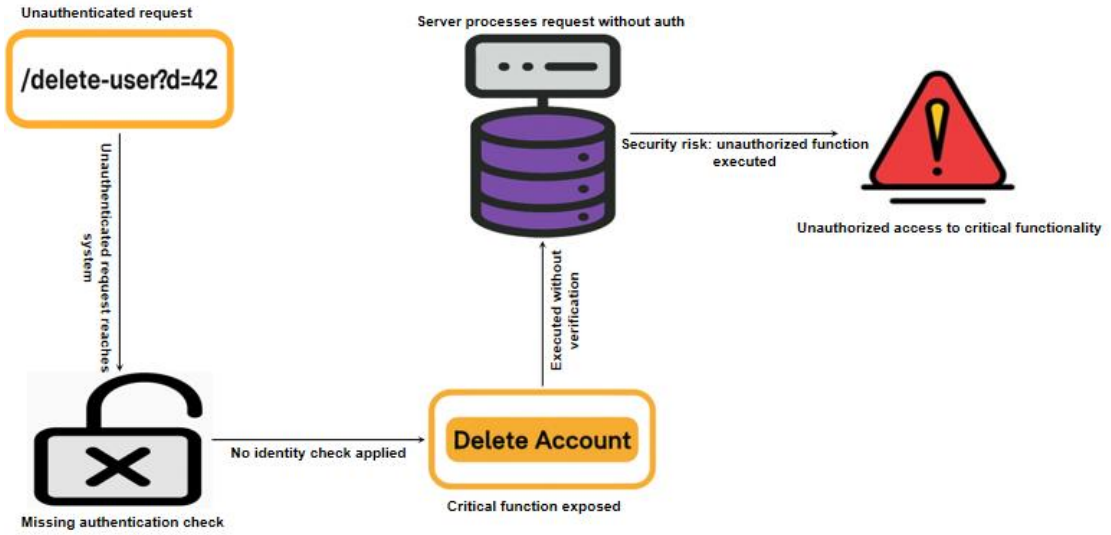

Figure 7:Illustration of a Missing Authentication for Critical Function Vulnerability

1.2.8. SQL injection:

SQL injection (SQLi) is a widely known yet persistently prevalent security vulnerability in web applications. It arises when user-supplied input is directly incorporated into SQL queries without proper validation or sanitization. This allows attackers to inject specially crafted SQL code into the query string, potentially altering its logic and bypassing authentication mechanisms. As a result, the database may execute unauthorized commands, exposing sensitive information or allowing full access to user data[13].

Figure 8 depicts an SQL Injection vulnerability, which occurs when untrusted input is concatenated directly into a SQL query without proper validation or sanitization. In this example, the application receives user input (e.g., from a login form) and embeds it into a SQL statement meant to retrieve a specific user record. However, due to the absence of input handling safeguards, an attacker can manipulate the input e.g., injecting ' OR 1=1 -- to alter the query logic. This results in the query returning the entire user dataset rather than a single authenticated user, effectively bypassing access controls. The figure highlights how such vulnerabilities can lead to unauthorized access to sensitive data, authentication bypass, or even full database compromise, depending on the privileges of the database user and the context of execution.

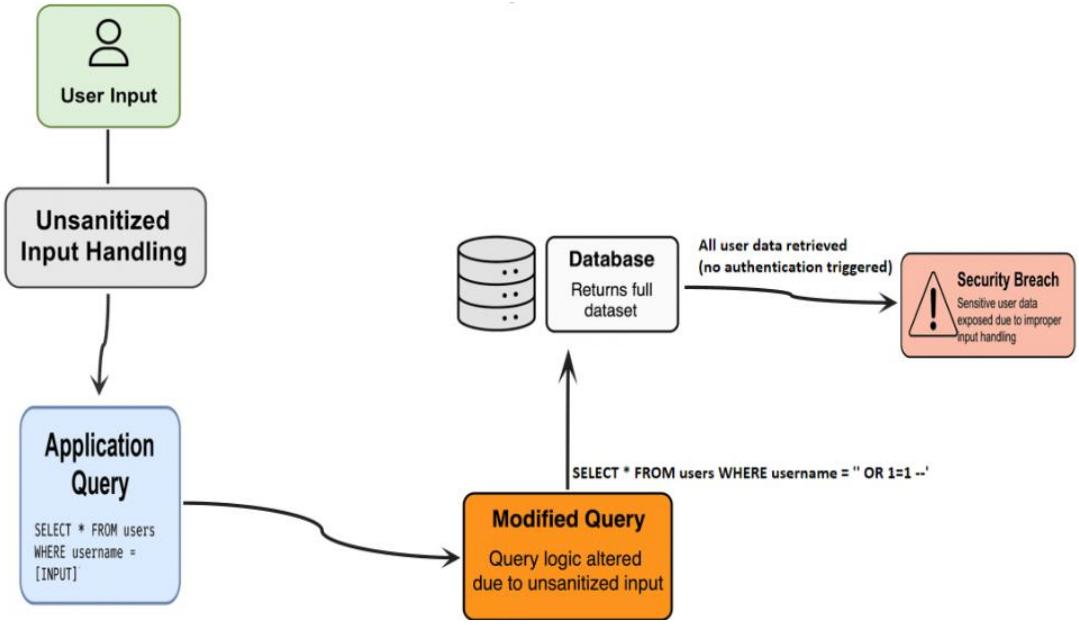

Figure 8:Illustration of an SQL Injection Vulnerability Exploiting Unsanitized User Input

1.2.9. Use After Free:

Use-after-free refers to the act of referencing memory after it has been freed. Exploiting such vulnerabilities may lead to program crashes, arbitrary code execution, or privilege escalation[14].

Figure 9 illustrates a Use-After-Free vulnerability, a type of memory corruption bug that occurs when a program continues to use a pointer to memory that has already been deallocated. In this scenario, memory is first allocated for an object and later released using free(object). However, the pointer referencing the released memory is not cleared or updated, resulting in what is known as a dangling pointer. If this pointer is subsequently used either unintentionally or via attacker control the program accesses freed memory that may have been reassigned to other data, leading to unpredictable behavior. As shown in the figure, this can result in corrupted memory access, which may be exploited to execute arbitrary code, cause a system crash, or leak sensitive information. This vulnerability is especially severe in low-level languages like C or C++ that provide direct memory management.

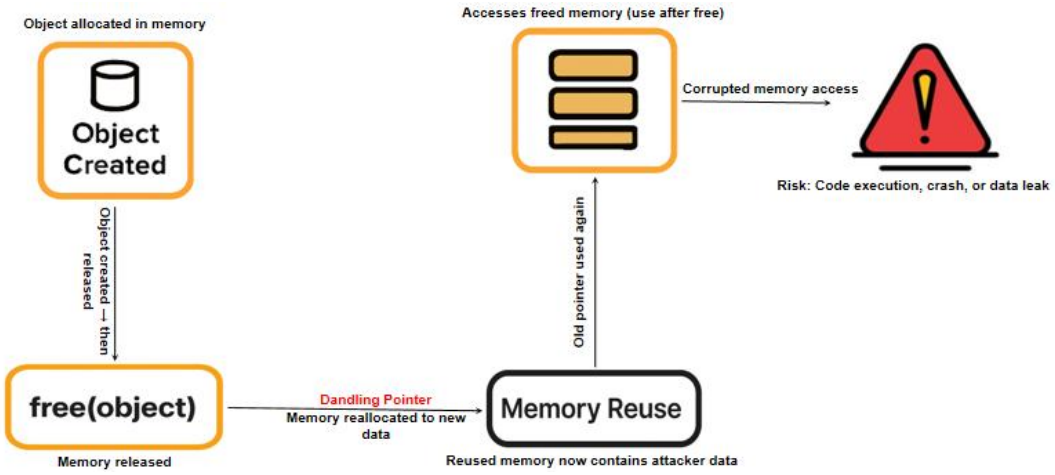

Figure 9:Illustration of a Use-After-Free Vulnerability Leading to Memory Corruption

1.2.10. Command Injection:

Command injection occurs when an attacker can inject system commands through application input, exploiting flaws in command construction logic to execute arbitrary commands[8].

Figure 10 depicts a Command Injection vulnerability, where user-supplied input is concatenated into a system command without proper validation. In this scenario, the application dynamically constructs a shell command (e.g., ping + user input) and executes it using system-level functions. Due to the lack of input sanitization, an attacker can manipulate the input to inject arbitrary commands, which are then executed by the shell. As the figure shows, this can result in full system access, remote code execution (RCE), or critical data loss, especially if the command is executed with high privileges. This type of vulnerability is particularly dangerous in web applications or scripts that call operating system utilities based on user input, emphasizing the need for strict validation and use of safe APIs.

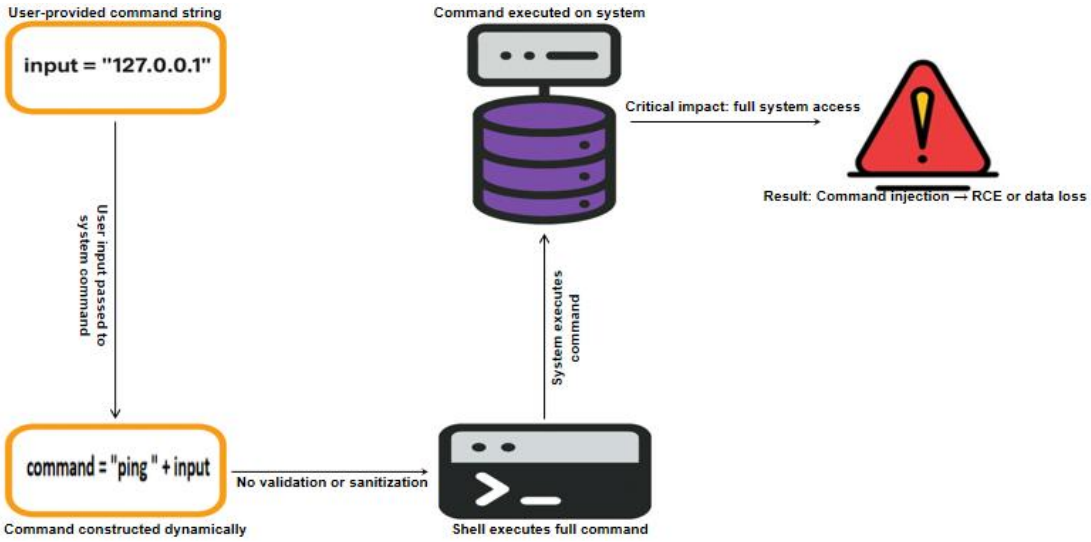

Figure 10:Illustration of Command Injection via Unsanitized Input in Dynamically Constructed Shell Commands

1.3. Vulnerability Detection Techniques:

1.3.1. Static analysis:

Static analysis refers to the examination of source code without executing the program. It represents a fundamental and proactive approach to identifying security vulnerabilities early in the software development lifecycle. By analyzing program structure, control flow, and data propagation, static analysis tools can detect potential flaws such as buffer overflows or improper input handling before the code is deployed[15]. For example, the unguarded use of functions like strcpy() in C can be flagged as a potential overflow condition[3]. These tools are valued for their speed, scalability, and suitability for automation, making them especially effective in large-scale codebases. However, they often produce a significant number of false positives,

which necessitates manual verification[16]. Studies by Wang and Guo (2021) and Jovanovic et al. (2006) emphasize the effectiveness of static analysis when integrated early in the development process, helping to prevent vulnerabilities from reaching production systems [15][17].

1.3.2. Dynamic Analysis:

Dynamic analysis, unlike static analysis which examines code without execution, adopts a more hands-on approach it runs the program and observes its behavior [18]. This makes it particularly effective for identifying vulnerabilities that manifest only when the software is active and interacting with real input. One common technique is fuzzing, where the tool injects unexpected or malformed data into the application to observe whether it crashes, leaks memory, or behaves abnormally [19][20]. Another effective method is runtime monitoring, which can detect issues such as use-after-free bugs problems that are nearly impossible to identify through static inspection alone [20]. A key strength of dynamic analysis is its ability to reveal how the software behaves under real-world conditions [18]. However, this realism comes with tradeoffs: it is more time-consuming, resource-intensive, and may still miss vulnerabilities if some code paths are not executed during testing [18]. This limitation was effectively discussed by Zheng and Zhang (2013), who emphasized the importance of path-sensitive techniques to explore deeper and less obvious execution flows [18].

1.3.3. Formal Verification:

Formal verification takes the process a step further. It is not about scanning code or running tests it involves mathematically proving that a program behaves exactly as intended. This is achieved through techniques such as model checking and theorem proving, which help ensure that the system satisfies specific properties under all possible conditions. The key advantage is that when formal verification determines a system to be secure, it offers a very high level of confidence in its correctness. However, it is not a lightweight approach; it demands extensive knowledge of formal logic and requires substantial computational resources, particularly for large or complex systems[21].

Farahmandi and Alizadeh (2015) demonstrated that these techniques can be applied even to low-level components such as arithmetic circuits, reinforcing that formal methods are not merely theoretical they can be practical when applied appropriately [21].

1.3.4. AI and Machine Learning(ML):

Machine learning is rapidly emerging as a critical component in the field of vulnerability detection. Rather than relying solely on hardcoded rules, ML models learn from large datasets containing both vulnerable and non-vulnerable code samples [22]. These models are capable of identifying subtle patterns and generalizing across diverse codebases [5]. Some are trained to detect vulnerabilities directly from code syntax, while others rely on static or dynamic features as input [22]. In more advanced implementations, large language models (LLMs) are fine-tuned on vulnerability-labeled datasets to not only detect but also repair insecure code [23]. According to Harzevili et al. (2023), the application of deep learning and LLMs

contributes to reducing false positives and improving detection accuracy, particularly in cases where traditional tools are insufficient. Although these AI-driven approaches are still evolving, they represent a promising direction for the automation of secure software development [24].

1.4. Conclusion:

This chapter introduced the critical role of cybersecurity in modern systems and explored the most common software vulnerabilities, including memory-based and input-handling flaws. It also reviewed traditional detection methods static, dynamic, and formal and highlighted their limitations. Finally, it emphasized the need for intelligent, automated solutions to address evolving security challenges, setting the stage for advanced techniques discussed in the next chapter.

Chaptre 02

Language Models

2.1. Introduction:

Over the past decade, software security has become increasingly complex as software vulnerabilities have emerged among the most critical threats to digital infrastructure. The widespread reliance on free open-source libraries, interconnected computing environments, and persistent architectural patterns has expanded the attack surface, making vulnerability detection a vital priority in cybersecurity domains [25].

Traditional detection methods, such as static and dynamic analysis, have proven effective in identifying well-defined vulnerabilitiesparticularly memory flaws in C/C++ programs. However, they struggle to detect newer vulnerabilities that are semantic in nature and lack explicit signatures [2].

To address these limitations, machine learning (ML) and deep learning (DL) techniques have emerged as promising alternatives. These models are capable of learning complex patterns from large-scale security datasets, allowing them to detect a wider range of vulnerabilities beyond manually encoded rules [26].

In particular, large language models (LLMs) such as BERT, CodeBERT, and GPT have become leading tools in this field. With their ability to derive semantic representations from source code, LLMs can gain deep contextual understanding of syntax and logic, significantly advancing automated vulnerability detection [27].

2.2. Historical Timeline of Vulnerability Detection:

Over the years, vulnerability detection has advanced from traditional static analysis to deep learning and LLM-based approaches. Zitser et al. (2004) highlighted the limitations of static tools[3]. Li et al. (2018–2019) introduced VulDeePecker and SySeVR, combining semantic analysis with deep models like BLSTM and BGRU[25]. Zhou et al. (2019) proposed Devign using GNNs on code graphs[28]. Wang et al. (2021) released CodeT5, a Transformer model for code understanding[29]. Recently, Ferrag et al. (2023–2025) developed datasets and benchmarks like FormAI and CASTLE, integrating LLMs with formal methods for improved accuracy[30].

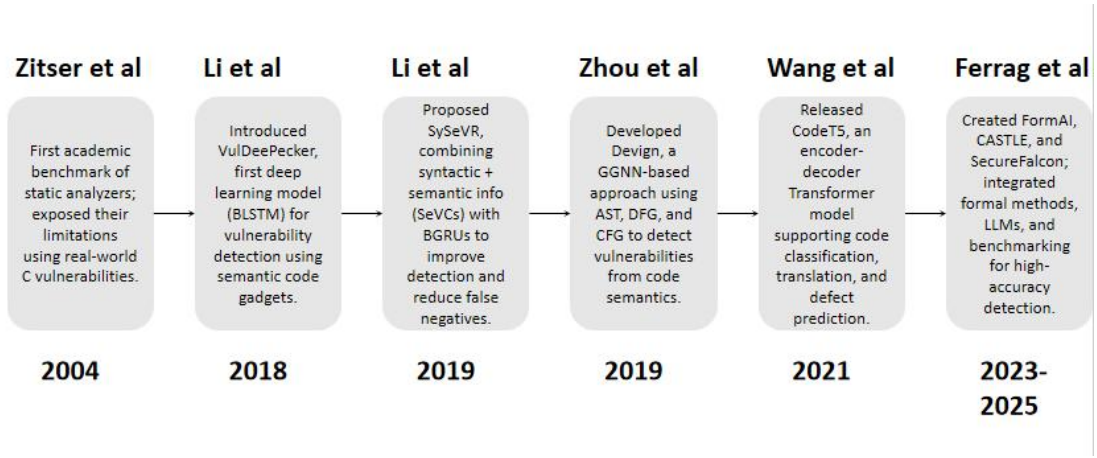

Figure 11:Timeline of key research milestones in vulnerability detection using static analysis, machine learning, and formal methods (2004–2025).

2.3. Machine Learning-Based Approaches:

Machine learning (ML) techniques have been extensively used in vulnerability detection due to their ability to learn patterns from labeled data and generalize to unseen code fragments. These models reduce the dependence on manually crafted rules and have achieved strong performance in various static code analysis tasks [31].

2.3.1. Feature Engenering:

Traditional machine learning (ML) pipelines for source code analysis rely heavily on hand-engineered features to represent code semantics and structure. Common features include code metrics (e.g., cyclomatic complexity, code churn), token frequency distributions, and structural information extracted from abstract syntax trees (ASTs). Additional elements such as API calls, variable naming patterns, and control flow structures are often selected to reduce feature dimensionality and improve model generalization [32].

2.3.2. Classification Algorithms:

After extracting feature vectors, traditional ML approaches utilize various classifiers such as Support Vector Machines (SVM), Random Forests, and Naive Bayes to determine whether a code segment is safe or vulnerable [32].

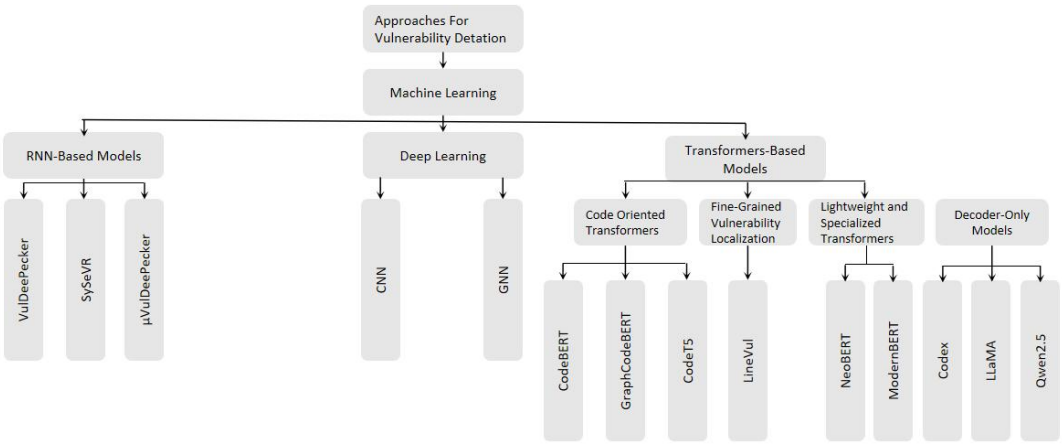

Figure 12:Taxonomy of Machine Learning and Transformer-Based Models for Vulnerability Detection

Figure 12 presents a taxonomy of machine learning-based models for software vulnerability detection. The classification is organized into three main categories: RNN-based models (e.g., VulDeePecker, SySeVR), deep learning models including CNN and GNN architectures, and transformer-based models. The transformer category is further subdivided into:

- ✓ Code-oriented Transformers (e.g., CodeBERT, GraphCodeBERT)
- ✓ Fine-grained vulnerability localization models (e.g., LineVul, CodeT5)
- ✓ Lightweight and specialized Transformers (e.g., NeoBERT, ModernBERT)
- ✓ Decoder-only LLMs (e.g., Codex, Qwen2.5)

This hierarchical organization is based on architectural design and the detection strategies employed by each model family.

2.4. RNN-Based Vulnearbility Detection:

Before the rise of transformer-based architectures, recurrent neural networks (RNNs) were among the first deep learning models applied to automated software vulnerability detection. Due to their sequential processing nature, RNNs were particularly well-suited for analyzing code written in low-level programming languages such as C and C++ [33].

2.4.1. VulDeePecker:

Introduced in 2018, VulDeePecker was the first deep learning-based system for software vulnerability detection using a bidirectional LSTM (BLSTM) architecture. The system introduced the concept of code gadgets, which are semantically connected lines of code centered around API or library calls. These gadgets were transformed into vector representations and fed into the neural network to classify vulnerable patterns [25].

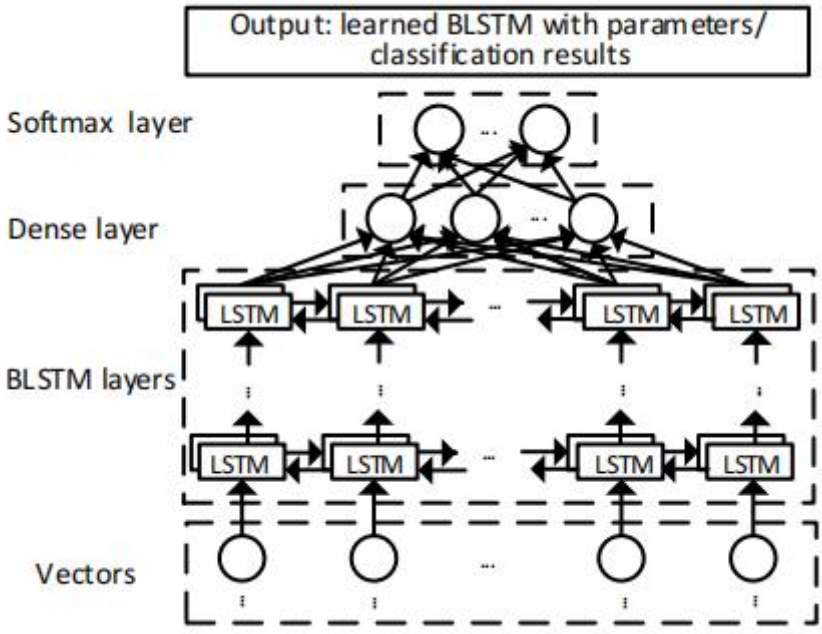

Figure 13:Internal Architecture of the BLSTM-Based Classifier in VulDeePecker[25]

The architecture of the VulDeePecker system, as illustrated in the figure, processes input vectors derived from code gadgets using stacked bidirectional LSTM (BLSTM) layers. These layers capture both forward and backward semantic context within the code. The resulting feature representations are then passed through a dense (fully connected) layer, followed by a softmax layer that outputs a classification indicating whether the code snippet is vulnerable or safe.

2.4.2. SySeVR:

SySeVR was introduced as an enhanced framework built upon VulDeePecker, aiming to improve detection accuracy and reduce false negatives. It achieves this by integrating both syntactic and semantic features of code into what are called Semantically Enriched Vulnerability Candidates (SeVCs). These enriched representations are processed using Bidirectional Gated Recurrent Units (BGRUs), which demonstrated superior performance in identifying diverse types of software vulnerabilities [33].

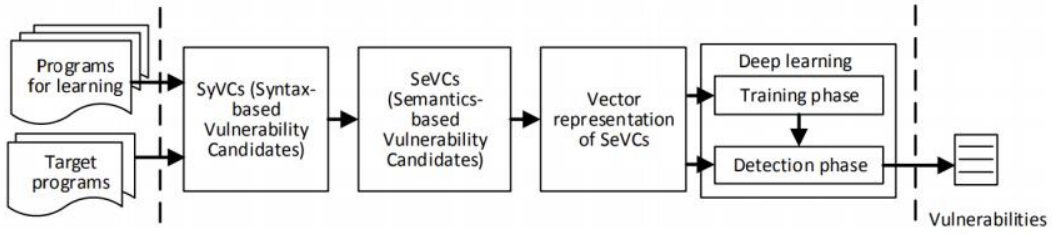

Figure 14:Overview of the SySeVR Framework for Deep Learning-Based Vulnerability Detection[33]

The SySeVR framework employs deep learning to detect software vulnerabilities through a structured multi-stage process. Initially, syntax-based vulnerability candidates (SyVCs) are extracted from source code. These are then enriched with semantic information to form Semantically Enriched Vulnerability Candidates (SeVCs). Once converted into vector representations, the SeVCs are fed into a deep learning model typically based on bidirectional recurrent networks which has been trained to identify patterns associated with code vulnerabilities.

2.4.3. µVulDeePecker:

µVulDeePecker was introduced to extend the capabilities of VulDeePecker by enabling multi-class vulnerability detection, rather than simple binary classification. The system enhances the original architecture through the integration of attention mechanisms and control-flow dependency tracking, allowing it to effectively capture features that distinguish between 40 different types of vulnerabilities. It achieved an impressive overall F1-score of 94.22%, significantly outperforming its predecessors in both accuracy and classification granularity [34].

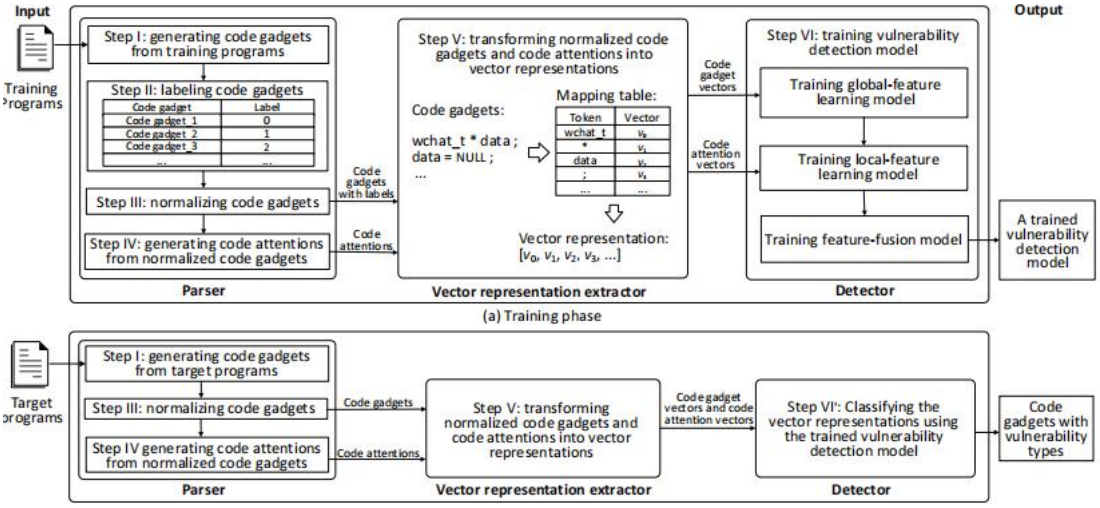

Figure 15:Architecture of µVulDeePecker Training and Inference Phases[34]

The complete architecture of the µVulDeePecker multi-class vulnerability detection system is illustrated in Figure 15. The framework consists of two main stages: training and inference.

During the training phase (top), code gadgets are extracted and standardized from labeled programs. To capture local semantic features, attention vectors are generated for each gadget. These vectors, along with the corresponding code tokens, are converted into numerical embeddings. Three specialized sub-models are then trained on these embeddings: a global-feature learner, a local-feature learner, and a fusion model that combines both representations to improve classification performance.

In the inference phase (bottom), the same processing pipeline is applied to unseen code samples. The trained model assigns a specific vulnerability type based on CWE identifiers to each code gadget, enabling fine-grained multi-class detection.

2.5. Deep Learning Models:

Deep learning has significantly reshaped the field of software vulnerability detection by allowing models to learn directly from raw source code, eliminating the need for manually engineered features. Unlike traditional machine learning approaches, deep models utilize hierarchical abstraction layers to automatically capture complex patterns associated with vulnerabilities.

Early research in this area explored the use of Convolutional Neural Networks (CNNs) and Deep Neural Networks (DNNs) to extract meaningful feature representations from source code. These models proved effective in identifying local patterns and token-level structures.

More recent advancements introduced Graph Neural Networks (GNNs), which are particularly well-suited for modeling the structural and semantic relationships embedded in programming languages, such as control flow, data flow, and abstract syntax trees.

2.5.1. CNN-Based Models:

Convolutional Neural Networks (CNNs) have been among the foundational deep learning architectures applied to software vulnerability detection. These models are particularly effective at capturing local patterns in code by processing structured input such as syntactic or semantic representations. By applying convolutional filters over code fragments, CNNs can identify fault-prone sections within functions and assist in pinpointing areas likely to contain vulnerabilities [33][35].

2.5.2. Graph-Based Models:

One of the most impactful advancements in deep learning for software vulnerability detection is the Devign model, proposed by Zhou et al. (2019). Devign leverages Gated Graph Neural Networks (GGNNs) to process complex code structures. It constructs a rich graph-based representation that integrates Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs). This allows the model to capture deeper semantic relationships compared to traditional sequence-based approaches.

The architecture of Devign consists of three core components [28]:

- Graph Embedding Layer: Builds heterogeneous program graphs from code elements.
- Recurrent Gated Graph Layers: Aggregate node-level information using gated message passing.
- Convolution Module: Performs graph-level classification to detect vulnerabilities.

2.6. Transformer-Based Models:

Transformer-based language models have significantly transformed the field of software vulnerability detection. By leveraging the self-attention mechanism, these models offer an effective way to capture the global context of source code, outperforming earlier methods based on recurrent and convolutional neural networks in both accuracy and scalability [36].

2.6.1. Code-oriented Transformers:

2.6.1.1. CodeBERT:

CodeBERT is a transformer-based model designed for code representation learning. It is pre-trained jointly on both natural language and source code from multiple programming languages. Using the masked language modeling (MLM) objective, CodeBERT is capable of handling a wide range of tasks, including classification, summarization, and security vulnerability detection. The model consists of 12 transformer encoder layers and learns bidirectional contextual embeddings to capture rich semantic information from code [37].

2.6.1.2. GraphCodeBERT:

GraphCodeBERT is an extension of CodeBERT that integrates data flow graphs (DFGs) into its training process. Through a graph-guided attention mechanism, the model is able to capture both syntactic structure and semantic relationships within the code. This enhancement enables GraphCodeBERT to better understand the dynamic behavior of programs by modeling how data flows through variables and functions during execution [37].

2.6.1.3. CodeT5:

CodeT5 is a Transformer-based model with an encoder-decoder architecture specifically designed for understanding and generating source code. Unlike encoder-only models (e.g., CodeBERT) or decoder-only models (e.g., CodeGPT), CodeT5 employs a unified text-to-text framework that allows it to handle a broader range of tasks. These include automated code repair, summarization, translation, and vulnerability classification.

A distinctive feature of CodeT5 is its identifier-aware pre-training strategy. In addition to predicting masked segments of code, the model is trained to recognize and recover key tokens such as variable and function names that often carry important semantic meaning. This significantly enhances its ability to perform tasks like clone detection and defect identification [38].

2.6.2. Fine-Grained Vulnerability Localization:

2.6.2.1. LineVul:

Many transformer-based models leverage self-attention mechanisms to assign importance scores to individual tokens within the source code. These attention

weights can then be aggregated to infer which lines of code are most likely to contain vulnerabilities. This approach not only enables fine-grained vulnerability localization but also captures the surrounding semantic context of the software flaw [39].

2.6.3. Lightweight and Specialized Transformers:

2.6.3.1. NeoBERT:

NeoBERT is an enhanced variant of the original BERT model, incorporating advanced optimization techniques such as Rotary Positional Encoding (RoPE) and FlashAttention. These architectural improvements result in significantly faster inference speed and reduced memory usage compared to models like ModernBERT. As a result, NeoBERT is well-suited for deployment in production-level vulnerability detection systems, where efficiency and scalability are critical [40].

2.6.3.2. ModernBERT:

ModernBERT is a lightweight and optimized version of the original BERT model, designed to enhance both training and inference efficiency while maintaining competitive classification accuracy. Although initially developed for natural language processing tasks in the medical domain specifically for Japanese radiology reports its architectural efficiency makes it a promising candidate for specialized applications such as software vulnerability detection [41].

2.6.4. Decoder-Only Models:

2.6.4.1. Codex:

Codex, also known as code-davinci-002, is a decoder-only transformer model developed by OpenAI and fine-tuned on large-scale code datasets collected from open-source repositories such as GitHub. As a direct successor to GPT-3 and a foundational model in the GPT-3.5 series, Codex supports a variety of code-centric tasks, including secure code generation, bug repair, and automated documentation.

Through its hybrid training on both natural language and billions of lines of source code, Codex enables more seamless interaction between programming intent and machine output, effectively bridging the gap between human instructions and executable code [42].

2.6.4.2. CodeLLaMA:

LLaMA (Large Language Model Meta AI) is a family of decoder-only transformer models developed by Meta as open-weight alternatives to proprietary large language models. These models offer scalable performance across a variety of natural language understanding and generation tasks. The second generation, LLaMA 2, includes models ranging from 7 billion to 70 billion parameters, and has rapidly gained prominence in both academic research and industry applications.

Among the LLaMA derivatives, LLaMA Guard stands out for its focus on safety and

security. It is a fine-tuned version of LLaMA 2 specifically trained for safety alignment, enabling it to classify and moderate both input prompts and model outputs based on a structured safety taxonomy. This includes the detection of high-risk content such as hate speech, violence, self-harm, and criminal intent making it particularly relevant for secure AI deployment scenarios [43].

2.6.4.3. Qwen2.5-Coder:

Qwen2.5-Coder is a specialized variant of the Qwen2.5 model family, designed for high-performance tasks related to source code, including code comprehension, bug fixing, and code generation. The model is well-suited for multilingual development environments, as it has been instruction-tuned using carefully curated datasets from nearly 40 different programming languages.

The training pipeline for Qwen2.5-Coder includes the following key steps [44]:

- Extraction of algorithmic code snippets from GitHub
- Generation of synthetic instruction–response pairs using data from developer Q&A forums
- Automated unit testing in a multilingual sandbox to validate correctness and ensure high performance across languages

2.7. Datasets and Benchmarks:

2.7.1. Big-Vul:

The Big-Vul dataset is one of the largest publicly available resources for C/C++ vulnerability analysis. It includes 3,754 labeled vulnerabilities collected from 348 open-source projects, covering 4,432 code commits mapped to 91 different CWE categories. Each record provides rich metadata, including the vulnerable function, the corresponding patch, commit information, and CVE/CWE identifiers, as well as the full code before and after the fix [45].

Big-Vul is particularly valuable due to its function-level granularity and inclusion of both vulnerable and patched code versions. These characteristics make it ideal for contrastive learning frameworks and fine-grained vulnerability localization models.

2.7.2. FormAI:

FormAI is a large-scale synthetic dataset specifically created for software vulnerability detection. It was generated using GPT-3.5-turbo and subsequently formally verified with ESBMC, ensuring logical consistency and correctness. The dataset contains over 112,000 compilable C programs, with 197,800 labeled vulnerabilities, each annotated with its corresponding CWE ID, line number, and function name [46].

A key advantage of FormAI lies in its foundation on formal verification,

which significantly reduces the occurrence of false positives and ensures high-quality ground-truth labeling. This makes it particularly effective for training models targeting memory-related vulnerabilities and fine-grained classification tasks.

2.7.3. CASTLE:

CASTLE (CWE Automated Security Testing and Low-Level Evaluation) is a manually curated benchmark composed of 250 compilable C micro-programs. Each test case is mapped to a specific Common Weakness Enumeration (CWE) category, allowing for line-level evaluation of both traditional static analysis tools and large language models (LLMs) [36].

A notable contribution of this benchmark is the introduction of the CASTLE Score a composite metric that evaluates detection performance across multiple dimensions, including true positives, false positives, and the criticality level of CWEs.

Since CASTLE is based on formal verification, it offers high-quality ground-truth labels and helps minimize labeling noise, making it highly suitable for evaluating precision-critical vulnerability detection systems.

2.7.4. Limitations of Existing Datasets:

Despite their importance in advancing the field, many existing vulnerability detection datasets suffer from critical limitations that reduce the reliability and generalizability of trained models.

2.7.4.1. Synthetic Bias and Unrealistic Scenarios:

Artificial datasets, such as Juliet, are intentionally designed to include vulnerabilities for the purpose of unit testing, making them less representative of real-world software environments. This artificial structure can result in models that achieve strong performance on benchmark evaluations but fail to generalize effectively to complex, real-world codebases [36].

2.7.4.2. Severe Class Imbalance:

Most real-world datasets suffer from extreme class imbalance, containing a disproportionately large number of non-vulnerable code samples. For instance, Draper VDISC and Big-Vul include a significant portion of safe code, making it difficult to train effective classifiers without applying techniques such as reweighting or oversampling [47].

2.7.4.3. Labeling Noise and Static Analysis Dependence:

Some datasets rely on automated static analysis tools, such as Cppcheck and Clang, to label vulnerabilities. While these tools are generally effective, they can introduce false positives and labeling noise, especially in datasets like Draper VDISC and SVCP4C, where labels may reflect tool behavior rather than actual vulnerabilities [47].

2.7.4.4. Duplication and Data Leakage:

Many vulnerability datasets contain a significant amount of duplicated code, which can result in data leakage between training and test sets. For example, Draper VDISC was found to include over 26% near-duplicate functions even after applying deduplication techniques, raising concerns about the validity of performance evaluation [47].

2.8. Benchmarks and Evaluation Metrics:

Effective comparison of software vulnerability detection tools requires the use of standardized benchmarks and clear performance metrics. In recent years, several benchmarks have been introduced to enable consistent, reproducible, and realistic evaluation of static analyzers, formal verification methods, and LLM-based models.

2.8.1.1. CASTLE Benchmark and Score:

The CASTLE benchmark is a manually curated micro-benchmark suite comprising 250 small C programs, each designed by cybersecurity experts to represent 25 distinct CWEs. Each sample is compilable and contains exactly one vulnerability or none, which simplifies and standardizes the evaluation process [30].

To assess tool performance, CASTLE introduces the CASTLE Score, a comprehensive metric defined as follows:

- ✓ True Positives (TP): +5 points per correct vulnerability detection
- ✓ True Negatives (TN): +2 points per correct identification of safe code
- ✓ False Positives (FP): −1 point per incorrect vulnerability detection
- ✓ Bonus (B): Up to +5 additional points depending on the CWE's rank in the

[MITRE Top 25]

This scoring system penalizes noisy tools and rewards precision, particularly when detecting high-risk CWEs [30].

2.8.1.2. MTEB: Massive Text Embedding Benchmark:

Although initially developed for evaluating text embeddings, the Massive Text Embedding Benchmark (MTEB) has recently been extended to assess code models across several tasks. These include:

- ➢ Semantic Code Search
- ➢ Code Summarization
- ➢ Code Retrieval

The English subset of MTEB comprises 7 tasks across 56 datasets, providing a comprehensive testing ground for embedding quality. Evaluation is based on cosine

similarity between token-level embeddings, and high performance typically requires contrastive fine-tuning [40].

2.8.1.3. Line-Level vs Function-Level Metrics:

Most existing datasets and models evaluate performance at the function level, typically treating vulnerability detection as a binary classification problem (vulnerable vs. safe). However, recent datasets and approaches such as LineVul and FormAI support line-level classification, which offers more granular insights and is better suited for real-world triage scenarios [36][48].

The most widely used evaluation metrics include [5]:

- Precision = TP / (TP + FP)
- Recall = TP / (TP + FN)
- F1-Score = 2 × (Precision × Recall) / (Precision + Recall)
- Accuracy = (TP + TN) / (TP + FP + TN + FN)

2.9. Use In Industry:

2.9.1. GitHub Copilot:

Powered by OpenAI's Codex, GitHub Copilot assists developers by offering real-time code suggestions within their development environment. However, several studies have shown that, without careful review, the generated code may introduce security vulnerabilities [49].

2.9.2. Microsoft Security Copilot:

Microsoft's Security Copilot is the first generative AI solution designed specifically for enterprise-level cybersecurity. It assists Security Operations Center (SOC) teams by automating key tasks such as threat detection, malware code analysis, and incident summarization, all powered by large language models (LLMs) [1].

2.9.3. NIST AI RMF:

In July 2024, the National Institute of Standards and Technology (NIST) released a specialized profile for Generative AI, emphasizing the need to incorporate AI trustworthiness principles including explainability, robustness, and provenance into secure software development practices [50].

[1] https://aka.ms/CopilotForSecurity

2.9.4. SAST and LLM-Augmented Pipelines:

Recent solutions have started integrating Static Application Security Testing (SAST) tools with large language models (LLMs) such as in the LSAST framework to enhance vulnerability detection accuracy. In these setups, the LLM processes the output of the SAST tool to identify potential vulnerability patterns that may have been overlooked by traditional scanners [51].

2.10. Related Work:

Table 1 provides an overview of recent studies focused on vulnerability detection using Large Language Models (LLMs). It summarizes the programming languages targeted, the datasets used, the number of samples, the detection methods applied, and whether the approaches support multiclass classification.

Name	Year	Lang	Dataset	Samples	Method	Multiclass
Tihanyi et al[52]	**2023**	**C**	**FormAI**	**50,000**	**ESBMC AI**	**Yes**
Sultan et al[37]	**2024**	**C/C++**	**Draper VDISC**	**24,492**	**LLM(DeLLNeuN)**	**NO**
Zhang et al[53]	**2024**	**C/C++**	**Real World Codes**	**223**	**LLM**	**Yes**
Shestov et al[54]	**2024**	**JAVA**	**Java Dataset**	**/**	**Finetuning +LLM**	**NO**
Tamberg et al[5]	**2025**	**JAVA**	**Java Julit 1.3**	**/**	**LLM+Static Tools**	**Yes**
Ferrag et al[46]	**2025**	**C/C++**	**FormAI +FalconVulnBD**	**243,075**	**Fine Tuned Transformer**	**Yes**
Dubniczky et al[30]	**2025**	**C**	**CASTLE**	**250**	**Formal Verification +prompted LLMs**	**Yes**

Table 1:Recent Studies on Vulnerability Detection Using LLMs

From this table, we observe a variety of approaches leveraging LLMs, sometimes in combination with static analysis tools or formal verification techniques. The datasets

used differ significantly in scale and origin, ranging from synthetic benchmarks to real-world code samples.

2.11. Conclusion :

This chapter reviewed key approaches in software vulnerability detection, from traditional ML methods to advanced LLM-based models. It also examined benchmark datasets and evaluation metrics, highlighting the trade-offs between precision, coverage, and real-world applicability. These insights lay the groundwork for the experimental analysis in Chapter 3.

Chaptre 03

Design and Performance Evaluation of an LLM-Based Software Vulnerability Detection System

3.1. Introduction:

As software systems grow in size and complexity, detecting security vulnerabilities at scale has become increasingly challenging. Traditional static and dynamic analysis techniques, while effective to some extent, often struggle with scalability, adaptability, and the accurate identification of subtle or novel vulnerability patterns. In response to these limitations, Large Language Models (LLMs)—pretrained on massive corpora of natural and programming languages—have emerged as powerful tools for semantic code understanding and classification tasks.

The motivation behind this chapter stems from the potential of LLMs to generalize across diverse vulnerability types, leverage contextual patterns in source code, and reduce reliance on handcrafted rules or manually engineered features. By fine-tuning transformer-based models such as BERT, CodeBERT, RoBERTa, DeBERTa, and NeoBERT, we aim to assess their effectiveness in multi-class vulnerability detection across real-world code samples.

This chapter outlines the experimental framework used to build and evaluate our proposed LLM-based detection system. It details the dataset selection and preprocessing pipeline, model training strategies, evaluation metrics, and comparison with both traditional machine learning baselines and other transformer-based models. The goal is to provide a comprehensive, reproducible, and fair assessment of LLMs in the context of CWE-based vulnerability classification.

3.2. Proposed LLM-Based Vulnerability Detector:

3.2.1. Overall System Architecture:

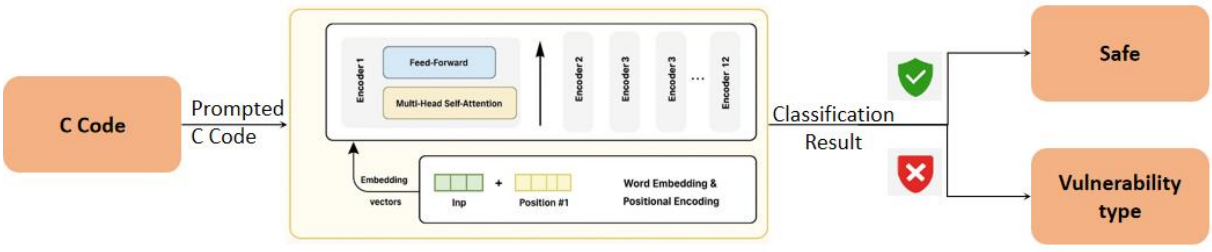

Figure 16:Overall System Architecture for Prompt-Based Vulnerability Classification using a

Transformer Model

The overall architecture of the vulnerability detection system is designed to classify C code snippets as either safe or containing a specific type of vulnerability. The system leverages a pre-trained transformer-based language model (e.g., BERT) fine-tuned for multi-class classification. The process involves several key components, as illustrated in Figure 16.

Step 1: Input Preparation – Prompted C Code

The raw C code is first processed and optionally augmented with a natural language prompt. This prompt is used to provide semantic guidance to the model. For example, prompt formats like:

“What type of vulnerability is this?” (Prompt A)

are prepended to the source code, forming the final text input. This allows the model to interpret the input in a more structured, question-aware context.

Step 2: Embedding and Encoding

The prompted code is tokenized and passed through an embedding layer, where each token is converted into a dense vector representation. Positional encodings are then added to retain sequence information.

The embedded sequence is then fed into the transformer encoder stack, composed of multiple layers of:

- ✓ Multi-head self-attention
- ✓ Feed-forward neural networks
- ✓ Layer normalization and residual connections

This architecture allows the model to learn long-range dependencies within the code, and understand control flow and data flow structures even across distant tokens.

Step 3: Classification

The output of the final encoder layer is pooled (typically using the [CLS] token representation) and passed through a classification head (usually a dense layer with softmax activation). The model predicts either:

- ➢ Safe – if the input contains no detectable vulnerability
- ➢ Vulnerable – if a vulnerability is detected, the model outputs the type of vulnerability (e.g., buffer overflow, NULL pointer dereference, etc.)

Step 4: Output Interpretation

The predicted label is interpreted as the final result. If vulnerable, the system identifies the most probable vulnerability category based on its training classes. Otherwise, the code is classified as safe.

3.2.2. Data Acquisition and Labeling:

3.2.2.1. Dataset Source: FormAI-v2

This study draws on the FormAI-v2 dataset a large-scale, high-quality benchmark meticulously crafted for the task of software vulnerability detection. It contains a total of 246,549 C-language code samples, each thoroughly annotated to reflect the presence or absence of security flaws. These samples were generated using a diverse set of advanced large language models (LLMs), including GPT-4, CodeLlama, Gemini Pro, and Falcon. To ensure neutrality and prevent bias, generation prompts followed a template-driven approach, deliberately avoiding the injection of known vulnerabilities while preserving variation in coding patterns.

3.2.2.2. Labeling Methodology:

Vulnerability annotations were applied using ESBMC (Efficient SMT-Based Bounded Model Checker), a formal verification tool that leverages symbolic model checking to assess the correctness and safety of C programs. When a flaw is detected, ESBMC outputs a counterexample trace a detailed execution path that confirms the violation. This labeling strategy grounds the dataset in formal semantics, providing a reliable and reproducible foundation for vulnerability classification.

Label Taxonomy: Safe vs. Vulnerable (CWE-Based)

Each sample in the dataset is assigned one of two primary labels:

- ✓ **Safe:** No security violations detected.
- ✓ **Vulnerable:** One or more security flaws identified.

For vulnerable samples, additional classification is provided using the Common Weakness Enumeration (CWE) framework, encompassing 42 unique vulnerability types. Representative examples include:

- **CWE-476:** Null Pointer Dereference
- **CWE-787:** Out-of-Bounds Write
- **CWE-190:** Integer Overflow
- **CWE-120:** Classic Buffer Overflow

Importantly, at least six of these CWE categories fall under the MITRE CWE Top 25, reinforcing the dataset's focus on high-impact, real-world vulnerabilities[54].

3.2.2.3. Dataset Characteristics:

- **Total code samples:** 246,549
- **Vulnerable samples:** ~63.47%
- **Diversity:** Some samples contain multiple CWE vulnerabilities
- **Common patterns:** Pointer misuse and memory safety errors (e.g.,

buffer overflows, null dereferencing) dominate the distribution

Thanks to its scale, structured design, and formal rigor, FormAI-v2 serves as an ideal benchmark for training and evaluating both LLM-driven detectors and traditional machine learning models in the realm of secure code analysis.

3.2.3. Preprocessing and Input Representation:

To prepare the raw C source code samples for effective training and inference with the BERT model, a structured preprocessing pipeline was implemented to ensure compatibility with the model's architecture while preserving the semantic integrity of each input.

The process began by filtering the dataset to retain only labeled samples, discarding entries with null or undefined values in the "Error type" column, as well as generic or ambiguous labels that did not correspond to specific vulnerability classes. From the cleaned dataset, a representative set of well-defined vulnerability types was selected based on label prevalence, ensuring that each class had sufficient support to contribute meaningfully to supervised learning and evaluation.

Each code snippet was then truncated to a maximum of 2,000 characters to fit within the tokenization limits of transformer models while retaining key semantic content. Following this, two distinct prompt formulations were designed to embed the source code within natural language contexts, allowing the model to leverage both code semantics and linguistic cues:

Prompt A:

Code:\n<code>\n\nQuestion: What type of vulnerability is this?

These prompted inputs were tokenized using the pretrained BERT tokenizer (bert-base-uncased), with padding and truncation enabled, and a maximum sequence length of 512 tokens applied. All inputs were processed in a function-agnostic manner, meaning each snippet was treated as a standalone unit, enabling the model to focus on localized patterns of vulnerability without segmenting the code by function or file.

Unlike purely supervised training with raw code tokens, this approach incorporated prompt engineering to explicitly encode the classification task within the input text. By integrating natural language cues into the input, the model was guided toward contextual understanding of code vulnerabilities through language, rather than relying solely on structural token patterns.

3.2.4. LLM Fine-Tuning Strategy:

All fine-tuning experiments were conducted using the Hugging Face Transformers library (version 4.35.2), with PyTorch as the computational backend. Training and evaluation were performed on Google Colab Pro utilizing an NVIDIA A100 GPU and a high-memory runtime environment (~83 GB RAM), which enabled efficient parallelization of batches and reduced training time.

Input tokenization was performed using the pretrained BERT tokenizer, with each sample truncated or padded to a maximum sequence length of 512 tokens. Tokenization was applied consistently across both Prompt A and Prompt B variants, preserving the structural integrity of code within a natural language context.

The training regime was standardized across all runs as follows:

- ✓ **Number of epochs:** 3
- ✓ **Batch size:** 16
- ✓ **Learning rate:** 2e-5
- ✓ **Optimizer:** AdamW (implicitly through Trainer)
- ✓ **Loss function:** CrossEntropyLoss
- ✓ **Weight decay:** 0.01
- ✓ **Warmup ratio:** 0.1
- ✓ **Evaluation strategy:** Per epoch using a stratified validation set
- ✓ **Mixed precision:** Disabled (fp16 = False) to maintain numerical stability

To ensure fair representation across classes and avoid bias toward dominant labels, a stratified 70/15/15 train-validation-test split was applied. Classes with fewer than two samples were excluded prior to training to prevent instability during optimization.

No prompt engineering or instruction tuning was applied. The model was trained directly on the tokenized input label pairs, enabling it to learn structural and semantic mappings between code patterns and CWE identifiers.

This fine-tuning strategy allowed BERT to effectively adapt its pretrained knowledge to the specific task of multi-class vulnerability classification, while preserving generalization capabilities and minimizing overfitting across frequent and rare CWE classes.

3.2.5. Inference Pipeline:

pipeline previously described, including prompt formatting, label encoding, and token length normalization. Once prompted and tokenized, each test sample was passed through the fine-tuned BERT classifier to obtain a raw logit vector representing the model's confidence over all predefined vulnerability categories.

These logits were subsequently processed using the softmax function to generate normalized probability scores. The final prediction was selected using the argmax of these probabilities, corresponding to the most likely CWE class or the "Safe" label. No threshold calibration or abstention mechanism was applied, although such strategies could be beneficial in security-sensitive deployments to reduce false positives or support uncertain classifications.

To facilitate evaluation and interpretability, the predicted numerical labels were decoded back to their original textual form using the fitted LabelEncoder. This enabled consistent generation of confusion matrices and classification reports. Additionally, the lightweight, prompt-based nature of the inference pipeline allows seamless integration into real-world tools such as static analyzers or CI/CD pipelines for automated vulnerability detection.

3.3. Experimental Setup and Methodology

3.3.1. Dataset Preparation:

To ensure efficient training and balanced evaluation, the dataset preparation process involved multiple filtering and transformation steps. From the original FormAI-v2 dataset containing over 246,000 C code samples, only entries with non-null vulnerability labels (i.e., valid entries in the "Error type" column) were retained. To reduce computational demands while maintaining class diversity, a random 50% sample of the labeled data was selected.

To address the issue of class imbalance, vulnerability classes with fewer than two occurrences were removed. This step helped avoid instability during training and ensured that each remaining class had sufficient representation. The top 9 most frequent CWE classes were then identified, and combined with the “Safe” label to form a 10-class classification task.

For all selected samples, the raw C code was truncated to 2,000 characters to preserve core semantics while fitting within the input constraints of transformer models. Each code sample was then processed using label encoding, where textual labels (e.g., “CWE-787”) were mapped to numerical class indices using LabelEncoder. This allowed for efficient handling of class targets during training.

Finally, the dataset was split into 80% training and 10% validation and 10% test subsets, using a stratified split to maintain the original class distribution across both sets. This ensured that the model would be exposed to a representative sample of each vulnerability type during both training and evaluation phases.

3.3.2. Fine-Tuning Environment:

All fine-tuning and training experiments were conducted on Google Colab Pro, utilizing an NVIDIA A100 GPU with High-RAM (~83 GB). This setup provided sufficient computational power and memory to support the large-scale fine-tuning of Transformer-based models, including NeoBERT. All training was performed in full 32-bit floating-point precision (fp16=False) to avoid instability or numerical errors during learning.

The deep learning pipeline was implemented using the Hugging Face Transformers library (v4.35.2) in conjunction with Accelerate (v0.25.0) and PyTorch as the backend.

Tokenization was handled using the pretrained tokenizer of each model, including NeoBERT's tokenizer, which applies subword-level encoding compatible with the architecture.

To maintain consistency and enable fair comparison across model architectures, we applied the same fine-tuning configuration to all Transformer-based models, including NeoBERT. The training regimen was as follows:

Batch size: 8

Learning rate: 2e-5

Epochs: 3

Loss function: CrossEntropyLoss

Data split: Stratified, to preserve class distribution across training and validation folds

For traditional machine learning baselines (e.g., Support Vector Machines, Random Forest, XGBoost), training was carried out using scikit-learn and XGBoost's GPU-accelerated backend to minimize training time while ensuring consistent evaluation under the same experimental conditions.

3.3.3. Evaluation Metrics:

To evaluate the effectiveness of the models, we employed a comprehensive suite of classification metrics designed to capture both overall performance and class-specific behavior:

Accuracy: Provided a high-level measure of performance by calculating the proportion of correct predictions across all classes.

Precision, Recall, and F1-Score: These core metrics were particularly useful for assessing model quality in the presence of class imbalance, helping quantify both false positives and false negatives.

Macro-Averaging: Treated all vulnerability classes equally, regardless of frequency, offering a balanced perspective on the model's ability to generalize across both common and rare CWE labels

Weighted (Micro) Averaging: Incorporated class distribution into the evaluation, highlighting potential biases by placing more emphasis on frequently occurring classes.

Confusion Matrices: Generated for each model to visualize classification outcomes, pinpoint misclassification trends, and identify specific CWE types that were frequently confused.

Per-Class Metric Breakdown: Reported individual precision, recall, and F1-score for each CWE class, allowing for a fine-grained analysis of detection strengths and weaknesses.

Altogether, these evaluation criteria offered a robust and nuanced view of each model's classification capabilities, with a focus on fairness and consistency across a diverse range of vulnerability categories.

For the traditional machine learning models we employed the same evaluation framework to ensure consistency and comparability across model types. All evaluations were conducted using the scikit-learn library.

Metrics such as accuracy, macro-averaged and weighted F1-scores, along with confusion matrices, were computed to assess overall and class-specific performance. To capture detailed per-class insights, we used scikit-learn's built-in classification_report function.

Although these models were not subjected to fine-tuning in the same way as the LLM-based systems, the evaluation protocol was kept uniform. This alignment allowed for a fair and objective comparison between traditional approaches and transformer-based architectures in terms of vulnerability detection capability.

3.4. Experimental Results:

3.4.1. Training & Validation Curves:

Interpretation of BERT Training and Validation Behavior:

BERT showed a clear learning trajectory across three epochs, with notable reductions in both training and validation loss (0.7328 → 0.5627 and 0.7249 → 0.6622, respectively). Performance metrics improved steadily, as accuracy rose to 75.53% and F1 reached 0.7334. These results reflect a stable training process and strong generalization, making BERT particularly suitable for capturing nuanced patterns in vulnerability classification tasks.

Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.732800	0.724977	0.745484	0.708894
2	0.673000	0.670745	0.753538	0.720295
3	0.562700	0.662251	0.755264	0.733448

Table 2:BERT Training and Validation Results

Interpretation of CodeBERT Training and Validation Behavior:

CodeBERT delivered a smooth and consistent optimization path, marked by a substantial drop in training loss (0.7198 → 0.5447) and corresponding improvements in validation loss. Accuracy climbed from 74.24% to 75.39%, while the F1 score advanced to 0.7370. Its ability to incrementally enhance both precision and recall across epochs highlights CodeBERT's strong representational capabilities for code-based vulnerability data.

Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.719800	0.718048	0.742377	0.719334
2	0.646000	0.673305	0.748015	0.720247
3	0.544700	0.668566	0.753883	0.736993

Table 3:CodeBERT Training and Validation Results

Interpretation of RoBERTa Training and Validation Behavior:

RoBERTa demonstrated gradual yet reliable improvement throughout training. The loss curves showed healthy declines (training: 0.7596 → 0.5912; validation: 0.7674 → 0.6839), while accuracy and F1 metrics consistently moved upward. Although its initial performance was lower than some peers, RoBERTa narrowed the gap by the final epoch, showcasing robust adaptability and learning efficiency over time.

Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.759600	0.767379	0.729720	0.706899
2	0.688900	0.701885	0.745369	0.714042
3	0.591200	0.683851	0.748015	0.729521

Table 3:RoBERTa Training and Validation Results

Table

Interpretation of DeBERTa Training and Validation Behavior:

DeBERTa stood out for its smooth convergence and well-aligned training dynamics. Loss values decreased predictably across epochs, and evaluation scores improved from 73.64% to 75.17% (accuracy) and from 0.7024 to 0.7305 (F1). The model displayed dependable performance gains without signs of volatility, underscoring its effectiveness in modeling both surface-level and contextual signals for vulnerability detection.

Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.736800	0.749636	0.736394	0.702435
2	0.676700	0.682836	0.743643	0.714103
3	0.568700	0.656114	0.751697	0.730455

Table 4:DeBERTa Training and Validation Results

Interpretation of NeoBERT Training and Validation Behavior:

NeoBERT displayed consistent and efficient learning progression, with the training loss decreasing from 0.6950 to 0.5341, and the validation loss remaining low and stable between 0.6970 and 0.6528, indicating no signs of overfitting. Throughout the training epochs, the model's accuracy improved from 74.36% to 75.08%, while the F1-score increased from 0.7222 to 0.7369, reflecting balanced learning and robust generalization. These stable trends highlight NeoBERT's effectiveness in capturing both structural and semantic patterns relevant to vulnerability detection.

Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.695000	0.696975	0.743643	0.722223
2	0.633400	0.652677	0.749971	0.726456
3	0.534100	0.652796	0.750892	0.736940

Table 5:NeoBERT Training and Validation Results

3.4.2. Test-Set Performance:

To evaluate model performance, we adopted a comprehensive set of classification metrics, including accuracy, macro-averaged F1, weighted F1, and a per-class breakdown.

BERT: The BERT model achieved a strong test-set accuracy of 76%, with a macro F1-score of 0.46 and a weighted F1-score of 0.73. It demonstrated high effectiveness on dominant classes such as buffer overflow on scanf (F1: 0.85) and NULL pointer dereference (F1: 0.79), reflecting its strength in modeling frequent patterns. However, performance declined notably on rare CWE types like ieee_div (F1: 0.19) and overflow on add (F1: 0.22). The gap between macro and weighted F1 indicates a class imbalance effect, where BERT performs best on well-represented vulnerabilities but struggles with low-frequency or semantically subtle ones.

	precision	recall	f1-score	support
arithmetic overflow on add	0.37	0.16	0.22	244
arithmetic overflow on floating-point ieee_div	0.50	0.12	0.19	92
arithmetic overflow on floating-point ieee_mul	0.44	0.18	0.26	144
arithmetic overflow on mul	0.38	0.17	0.23	207
arithmetic overflow on sub	0.39	0.15	0.21	305
buffer overflow on fscanf	0.51	0.47	0.49	74
buffer overflow on scanf	0.81	0.91	0.85	4256
dereference failure: NULL pointer	0.78	0.80	0.79	1578
dereference failure: array bounds violated	0.70	0.73	0.71	1292
dereference failure: invalid pointer	0.66	0.61	0.63	499
accuracy			0.76	8691
macro avg	0.55	0.43	0.46	8691
weighted avg	0.73	0.76	0.73	8691

Table 6:Classification Report for BERT

CodeBERT: CodeBERT delivered slightly stronger generalization, with a macro F1-score of 0.47 and weighted F1 of 0.74, alongside a 75% accuracy. It achieved the highest F1 on buffer overflow on scanf (0.86) and consistently strong results on pointer-related errors such as array bounds violation (F1: 0.71). Its performance on ieee_mul (F1: 0.26) and other arithmetic-related errors was slightly better than RoBERTa and BERT, suggesting improved handling of numeric expressions. These results confirm that CodeBERT benefits from its training on code-specific data, offering more robust detection across structural patterns in C source code.

	precision	recall	f1-score	support
arithmetic overflow on add	0.38	0.20	0.26	244
arithmetic overflow on floating-point ieee_div	0.52	0.14	0.22	92
arithmetic overflow on floating-point ieee_mul	0.48	0.20	0.28	144
arithmetic overflow on mul	0.42	0.21	0.28	207
arithmetic overflow on sub	0.36	0.22	0.27	305
buffer overflow on fscanf	0.48	0.43	0.45	74
buffer overflow on scanf	0.82	0.90	0.86	4256
dereference failure: NULL pointer	0.77	0.80	0.79	1578
dereference failure: array bounds violated	0.69	0.74	0.71	1292
dereference failure: invalid pointer	0.67	0.55	0.60	499
accuracy			0.75	8691
macro avg	0.56	0.44	0.47	8691
weighted avg	0.73	0.75	0.74	8691

Table 7:Classification Report for CodeBERT

RoBERTa: RoBERTa yielded similar results to BERT, with an overall accuracy of 75%, macro F1-score of 0.46, and weighted F1-score of 0.73. It maintained high predictive strength for CWE classes with larger support, such as buffer overflow on scanf (F1: 0.84) and NULL pointer dereference (F1: 0.78). However, its performance on underrepresented arithmetic overflow types remained limited, with F1-scores dropping to as low as 0.19. This demonstrates that while RoBERTa is reliable for

standard cases, it shares similar vulnerabilities to BERT when confronted with rare or ambiguous code structures.

	precision	recall	f1-score	support
arithmetic overflow on add	0.37	0.19	0.25	244
arithmetic overflow on floating-point ieee_div	0.50	0.12	0.19	92
arithmetic overflow on floating-point ieee_mul	0.44	0.24	0.31	144
arithmetic overflow on mul	0.34	0.17	0.23	207
arithmetic overflow on sub	0.40	0.18	0.25	305
buffer overflow on fscanf	0.48	0.43	0.46	74
buffer overflow on scanf	0.81	0.89	0.85	4256
dereference failure: NULL pointer	0.76	0.80	0.78	1578
dereference failure: array bounds violated	0.69	0.73	0.71	1292
dereference failure: invalid pointer	0.66	0.56	0.60	499
accuracy			0.75	8691
macro avg	0.55	0.43	0.46	8691
weighted avg	0.72	0.75	0.73	8691

Table 8:Classification Report for RoBERTa

DeBERTa: DeBERTa reached an accuracy of 75%, with macro F1 of 0.45 and weighted F1 of 0.73. It performed comparably to other models on major vulnerability types, particularly buffer overflow on scanf (F1: 0.85) and NULL pointer dereference (F1: 0.77). Nonetheless, it exhibited a slightly steeper drop in performance on arithmetic and floating-point errors, where F1-scores fell below 0.26 for most classes. Despite its smooth training dynamics, the test-set results indicate DeBERTa may require further tuning or context expansion to improve its sensitivity to rare CWE categories.

	precision	recall	f1-score	support
arithmetic overflow on add	0.40	0.16	0.22	244
arithmetic overflow on floating-point ieee_div	0.46	0.13	0.20	92
arithmetic overflow on floating-point ieee_mul	0.53	0.11	0.18	144
arithmetic overflow on mul	0.39	0.20	0.26	207
arithmetic overflow on sub	0.36	0.18	0.24	305
buffer overflow on fscanf	0.51	0.38	0.43	74
buffer overflow on scanf	0.81	0.90	0.85	4256
dereference failure: NULL pointer	0.76	0.81	0.79	1578
dereference failure: array bounds violated	0.68	0.74	0.71	1292
dereference failure: invalid pointer	0.68	0.57	0.62	499
accuracy			0.75	8691
macro avg	0.56	0.42	0.45	8691
weighted avg	0.73	0.75	0.73	8691

Table 9:Classification Report for DeBERTa

NeoBERT: NeoBERT achieved an accuracy of 75%, with a macro F1-score of 0.45 and a weighted F1-score of 0.74, placing it on par with other transformer-based models. It showed strong performance on high-frequency vulnerability types such as

buffer overflow on scanf (F1: 0.85) and NULL pointer dereference (F1: 0.78), indicating reliable learning on dominant CWE classes. However, similar to other models, NeoBERT struggled with rare categories, particularly arithmetic overflows and floating-point operations, where F1-scores remained around or below 0.30. These results suggest that despite stable training behavior, further refinement—such as data augmentation or enhanced architectural context—may be needed to improve NeoBERT's recall on underrepresented vulnerabilities.

	precision	recall	f1-score	support
arithmetic overflow on add	0.36	0.22	0.27	244
arithmetic overflow on floating-point ieee_div	0.35	0.14	0.20	92
arithmetic overflow on floating-point ieee_mul	0.48	0.22	0.30	144
arithmetic overflow on mul	0.36	0.25	0.30	207
arithmetic overflow on sub	0.38	0.23	0.29	305
buffer overflow on fscanf	0.51	0.50	0.51	74
buffer overflow on scanf	0.82	0.89	0.85	4256
dereference failure: NULL pointer	0.77	0.80	0.79	1578
dereference failure: array bounds violated	0.68	0.74	0.71	1292
dereference failure: invalid pointer	0.67	0.53	0.59	499
accuracy			0.75	8691
macro avg	0.54	0.45	0.48	8691
weighted avg	0.73	0.75	0.74	8691

Table 10:Classification Report for NeoBERT

3.4.3. Confusion Matrix & Error Analysis:

BERT: The confusion matrix of the BERT model reveals notable patterns of misclassification among semantically related CWE categories. While the model achieved strong diagonal values for high-frequency classes like "buffer overflow on scanf" (3870 correctly predicted instances) and "dereference failure: NULL pointer" (1264), it also showed systematic confusion in several areas.

Arithmetic overflows were especially prone to mutual misclassification. For instance, instances of "arithmetic overflow on add" were often confused with "arithmetic overflow on sub" (134 misclassifications) and "NULL pointer dereference" (30 misclassifications), highlighting difficulties in distinguishing numerical operations with similar syntax patterns. Similarly, floating-point operations, such as ieee_mul and ieee_div, were occasionally mislabeled as generic arithmetic overflows or misrouted toward "buffer overflow on scanf", likely due to token overlap.

Pointer-related vulnerabilities, such as "dereference failure: invalid pointer", were frequently confused with "NULL pointer dereference" (117 times), and vice versa. This pattern suggests a challenge in recognizing subtle semantic cues, especially in snippets lacking strong contextual anchors like type annotations or initialization status.

In rare cases, misclassifications spilled over between logically distant classes. For example, "buffer overflow on fscanf" was wrongly identified as "scanf" (17 times) or even "NULL pointer dereference", indicating that shallow surface features may mislead the model in lower-frequency classes.

Overall, while BERT demonstrated strong discrimination power on dominant vulnerabilities, its errors expose persistent challenges in separating closely related CWE types. These findings underline the need for context-aware representations and potentially integrating static code analysis cues to reduce ambiguity in edge cases.

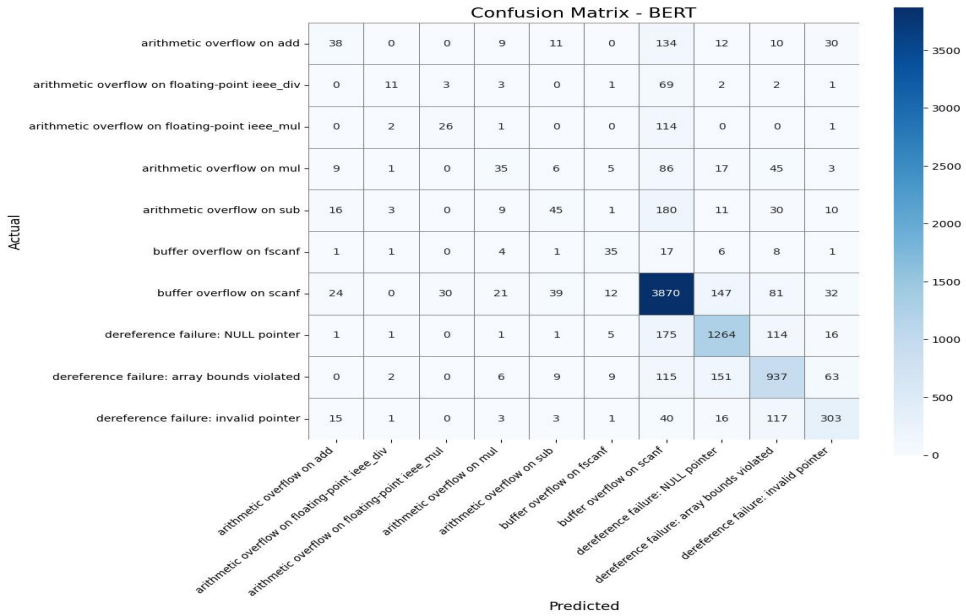

Figure 17:Confusion Matrix of BERT on the Top 10 Most Frequent CWE Vulnerability Classes

CodeBERT: The confusion matrix of CodeBERT reveals generally strong performance on frequent vulnerability classes, but also points to notable confusion between closely related categories. As expected, “buffer overflow on scanf” yielded the highest correct predictions (3819 instances), followed by “NULL pointer dereference” (1270 correct predictions), confirming the model’s effectiveness on dominant classes.

However, arithmetic-related errors continued to show high inter-class misclassification. For example, “arithmetic overflow on add” was frequently mistaken for “sub” (120 times) and “invalid pointer” (33 times), indicating semantic overlap in low-level arithmetic expressions. Similarly, “floating-point ieee_mul” saw significant misrouting toward “buffer overflow on scanf” (112 instances), reflecting a trend where low-confidence samples drift toward over-represented labels.

In pointer-related errors, “invalid pointer dereference” was misclassified as “NULL pointer” 118 times, while “array bounds violation” had 153 false positives as “NULL pointer” and 103 as “scanf”, reinforcing the model’s struggle to distinguish subtle memory access patterns when syntactic cues are ambiguous.

Overall, despite strong overall classification metrics, CodeBERT’s confusion matrix exposes weaknesses in separating semantically similar classes and suggests a tendency toward overgeneralization in the presence of limited context. Improvements could focus on enhancing code-level structural understanding or leveraging control/data flow features to reduce these ambiguities.

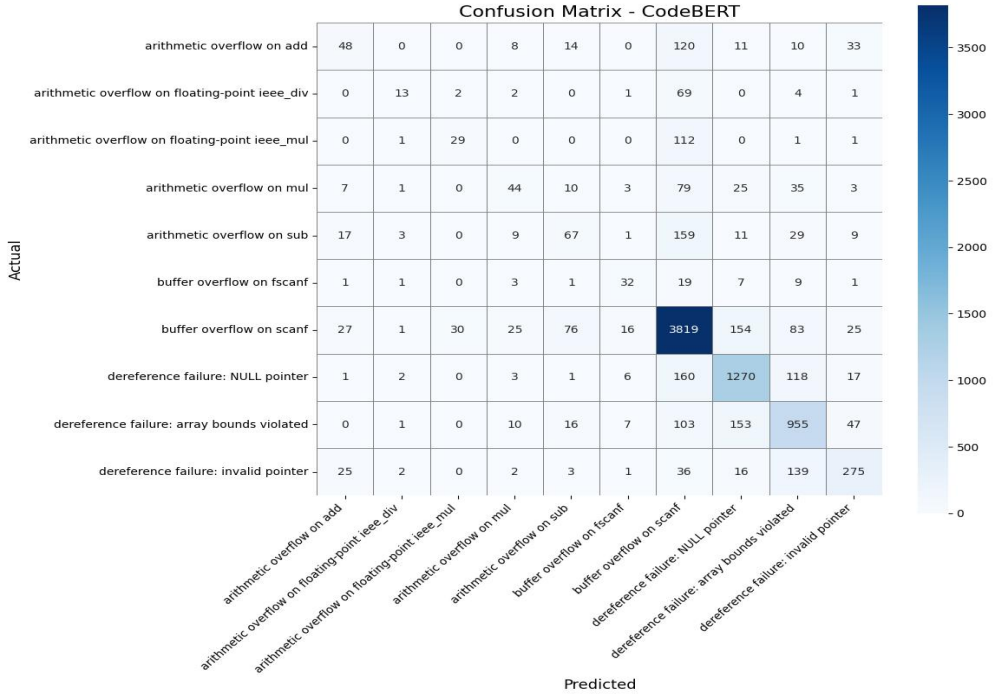

Figure 18:Confusion Matrix of CodeBERT on the Top 10 Most Frequent CWE Vulnerability Classes

RoBERTa: The confusion matrix of RoBERTa reveals misclassification trends similar to BERT, particularly in semantically adjacent classes. Despite RoBERTa's improved accuracy on several major vulnerability types, certain weaknesses persist in differentiating overlapping CWE categories.

The model excelled at classifying "buffer overflow on scanf", with 3802 correct predictions, and "NULL pointer dereference" with 1266 correct. However, it also frequently misclassified pointer dereference errors, such as "invalid pointer dereference", which was wrongly predicted as "NULL pointer" (137 times), and vice versa. This pattern reflects difficulty in discerning contextual differences when pointer types or memory regions are under-specified.

Arithmetic-related vulnerabilities also posed a challenge. For example, "arithmetic overflow on add" was confused with "sub" (128 times), while "ieee_mul" was often misinterpreted as a generic overflow type or routed to "buffer overflow on scanf". Such errors stem from shared lexical patterns and limited arithmetic differentiation in short code snippets.

RoBERTa also exhibited leakage of predictions into high-frequency classes. Misclassifications into "buffer overflow on scanf" appeared from nearly all other categories, showing a bias toward dominant labels in ambiguous scenarios.

In summary, while RoBERTa performed reliably on dominant CWE types, its confusion matrix exposes challenges in differentiating between structurally or semantically similar vulnerabilities. These findings point to the need for integrating

richer code semantics or control-flow context during training to reduce class confusion.

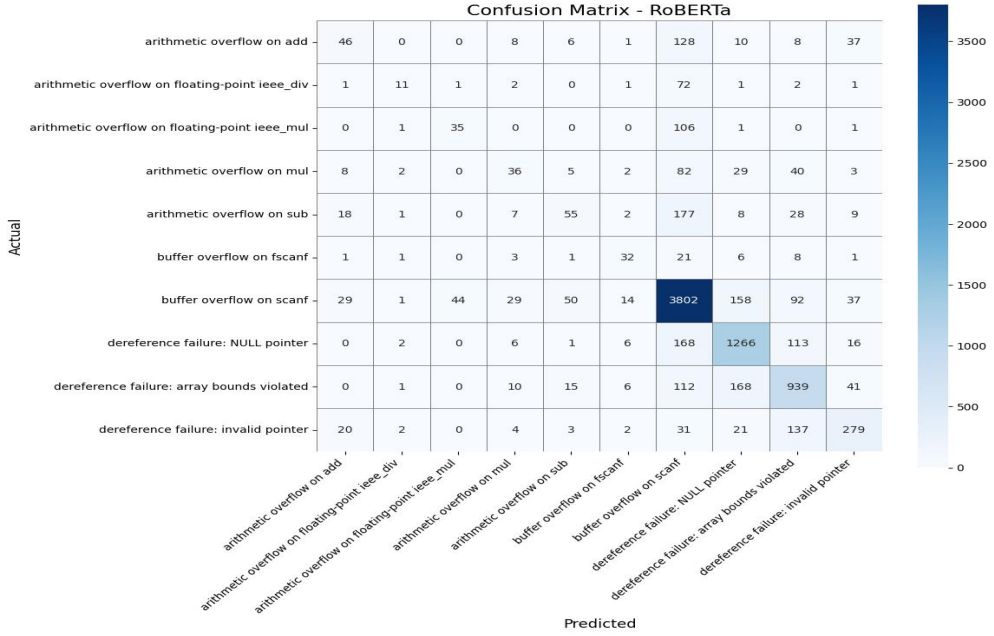

Figure 19:Confusion Matrix of RoBERTa on the Top 10 Most Frequent CWE Vulnerability Classes

DeBERTa: DeBERTa demonstrated strong accuracy on high-frequency vulnerability classes but continued to exhibit confusion in semantically overlapping categories. The model correctly identified 3827 instances of “buffer overflow on scanf” and 1277 for “NULL pointer dereference”, affirming its robust performance on dominant CWE labels.

Nonetheless, the matrix shows consistent misclassifications between arithmetic-related overflows. For example, “arithmetic overflow on add” was mistakenly labeled as “sub” 130 times and as “invalid pointer dereference” 35 times. Similarly, “floating-point ieee_mul” had over 125 false positives into “scanf”, suggesting that token-level similarity and limited contextual cues led the model to collapse rare types into more frequent categories.

In pointer-related vulnerabilities, “invalid pointer dereference” was misclassified as “NULL pointer” 137 times, and vice versa. The high confusion rates in this group suggest limitations in distinguishing between similar dereference failure modes without deeper control-flow awareness.

Moreover, low-confidence predictions tended to default into the overrepresented class “buffer overflow on scanf”, reinforcing a model bias toward high-support categories under ambiguity.

In conclusion, DeBERTa proved effective for major vulnerability classes but still faced ambiguity when distinguishing semantically similar CWE types. Addressing these issues may require incorporating richer code structure representations or training with contrastive strategies to better separate close-category features.

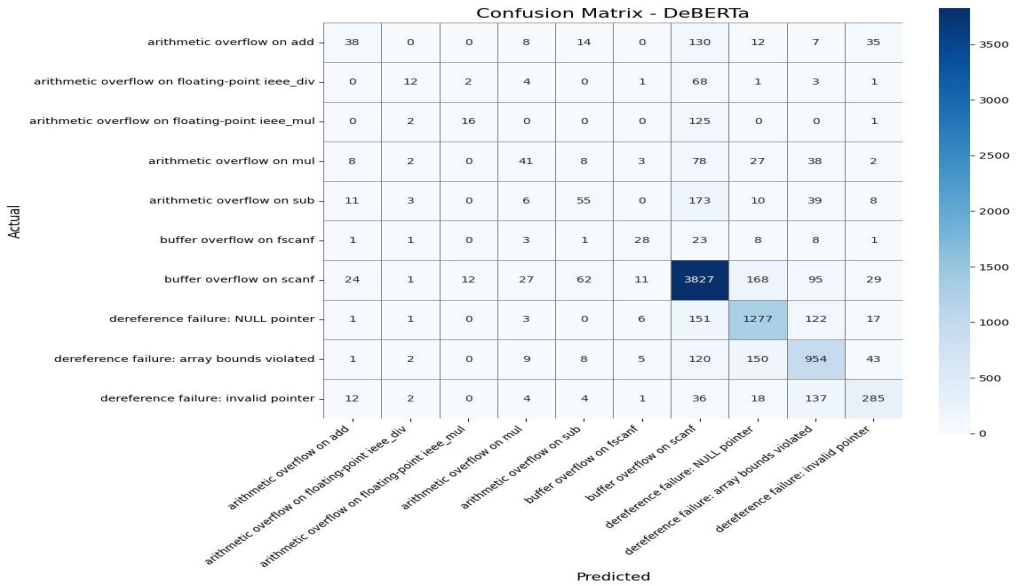

Figure 20:Confusion Matrix of DeBERTa on the Top 10 Most Frequent CWE Vulnerability Classes

NeoBERT: NeoBERT maintained strong classification performance on dominant vulnerability types while continuing to face challenges in distinguishing semantically similar CWE classes. According to the updated confusion matrix, the model correctly identified 3775 instances of “buffer overflow on scanf” and 1269 instances of “NULL pointer dereference”, confirming its robustness on frequent patterns.

However, arithmetic overflows remained a consistent source of misclassification. For example, “arithmetic overflow on sub” was often mislabeled as “add” (117 times) and “scanf” (29 times). Similar confusion appeared in floating-point operations, such as “ieee_mul”, which was incorrectly predicted as “scanf” over 100 times. These trends highlight the model’s difficulty in separating low-frequency categories that share lexical and structural patterns.

In the pointer-related group, “invalid pointer dereference” was misclassified as “NULL pointer” in 145 cases, reinforcing the pattern of overlap observed across models like BERT and DeBERTa. Additionally, there was a consistent funneling of ambiguous samples into high-frequency classes, especially “buffer overflow on scanf”, indicating a prediction bias caused by class imbalance.

Despite these challenges, NeoBERT’s overall confusion profile remains stable and in line with other LLM-based classifiers. Its performance suggests that while the model captures dominant classes effectively, improvements such as class-aware training strategies or enhanced code context modeling could reduce confusion across semantically adjacent vulnerabilities.

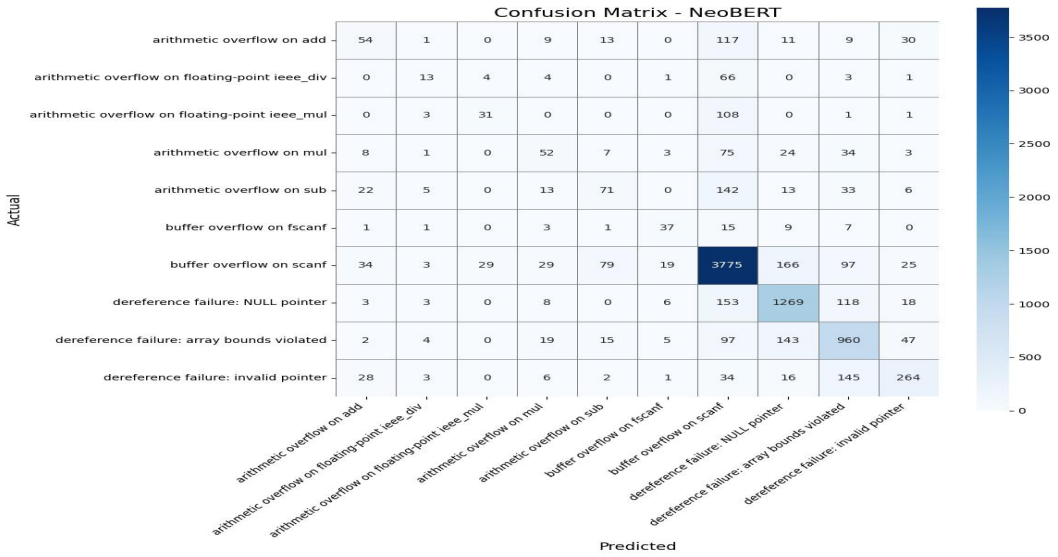

Figure 21:Confusion Matrix of NeoBERT on the Top 10 Most Frequent CWE Vulnerability Classes

3.4.4. Comparison with Baselines:

This section compares traditional machine learning models and large language models (LLMs) for multi-class software vulnerability detection. Table 16 summarizes their performance using key metrics: accuracy, macro F1, and weighted F1. This evaluation highlights how well each model handles class diversity and reveals their respective strengths and limitations.

Random Forest Classifier:

Vectorization: TF-IDF (char-level, n-gram 3–6).

Hyperparameters: 100 estimators (trees), default settings.

	Accuracy	F1 Score (Macro)	F1 Score (Weighted)
0	0.6301	0.1436	0.6071

Table 11:Test-Set Evaluation Results for Random Forest

Random Forest demonstrated solid performance in terms of accuracy and weighted F1-score, indicating that it was effective at capturing patterns within the more frequent vulnerability classes. However, its relatively low macro F1-score reveals a limitation: the model struggled to generalize across less represented (rare) CWE categories, suggesting a performance bias toward dominant classes in the dataset.

Logistic Regression:

Vectorization: TF-IDF (char-level, n-gram 3–5).

Hyperparameters: solver='saga', max_iter=300, tol=1e-3, parallel CPU execution.

	Accuracy	F1 Score (Macro)	F1 Score (Weighted)
0	0.6552	0.0221	0.5881

Table 12:Test-Set Evaluation Results for Logistic Regression

Logistic Regression achieved the second-highest accuracy among all evaluated models, yet it recorded the lowest macro F1-score, highlighting a pronounced bias toward majority classes. This result suggests that while the model handled frequent vulnerabilities reasonably well, it exhibited minimal predictive capability for rare or nuanced CWE categories. The poor generalization indicates that a linear decision boundary was inadequate for capturing the complex semantic patterns inherent in source code vulnerabilities.

XGBoost:

Vectorization: TF-IDF (char-level, n-gram 3–5), max_features=3000.

Hyperparameters: GPU-accelerated config with n_estimators=200, max_depth=6, learning_rate=0.1, gamma=1, tree_method="gpu_hist".

	Accuracy	F1 Score (Macro)	F1 Score (Weighted)
0	0.2663	0.0034	0.1839

Table 13:Test-Set Evaluation Results for XGBoost

XGBoost, despite being GPU-accelerated and configured with 200 decision trees, showed notably poor performance on this dataset. Both its macro and weighted F1-scores were among the lowest, indicating either severe overfitting or limited generalization capability likely influenced by class imbalance or suboptimal hyperparameter settings. The model struggled to make consistent predictions across most CWE categories, highlighting its sensitivity to the dataset's skewed label distribution and structural complexity.

Naive Bayes:

Vectorization: TF-IDF (char-level, n-gram 3–5), max_features=800.

Hyperparameters: alpha=0.1 (MultinomialNB).

	Accuracy	F1 Score (Macro)	F1 Score (Weighted)
0	0.5855	0.0294	0.529

Table 14:Test-Set Evaluation Results for Naive Bayes

Naive Bayes served as a fast and lightweight baseline, offering minimal computational overhead. Although it achieved an accuracy slightly above 58%, its macro and weighted F1-scores were notably low. The model's performance was hindered by its strong feature independence assumption, which fails to account for the structural and contextual dependencies present in source code. Given the complexity of programming syntax and semantics, this limitation significantly impacted its ability to detect diverse and subtle vulnerability patterns.

LinearSVC:

Vectorization: TF-IDF (char-level, n-gram 3–5), max_features=800.

Hyperparameters: C=1.0, max_iter=1000, dual=False.

	Accuracy	F1 Score (Macro)	F1 Score (Weighted)
0	0.6628	0.0644	0.591

Table 15:Test-Set Evaluation Results for SVC

Linear SVM achieved the highest overall accuracy among the traditional machine learning models. It also recorded the second-best macro F1-score, following Random Forest, which indicates a relatively stronger capacity to handle class imbalance. Nonetheless, its performance remained limited on rare CWE classes, as reflected in the confusion matrix, where several minority categories were consistently misclassified. Despite this shortcoming, Linear SVM emerged as the most balanced baseline within the traditional model group, offering a solid trade-off between generalization and class-specific sensitivity.

Task	ML Type	Model	Metric		
			Accuracy	Macro	Weighted
	Traditional ML	Random Forest	0.6301	0.1436	0.6071
		Logistic Regression	0.6552	0.0221	0.5881
		Naive Baise	0.5855	0.0294	0.529
		Linear	0.6628	0.0644	0.591

Multiclass Classification		**SVC**			
		XGBoost	0.2663	0.0034	0.1839
	LLMs	**BERT**	0.76	0.46	0.73
		CodeBERT	0.75	0.47	0.74
		RoBERTa	0.75	0.46	0.73
		DeBERTa	0.75	0.45	0.73
		NeoBERT	0.75	0.48	0.74

Table 16:Comparison of the Proposed Model with LLM Models and Traditional ML

3.5. Discussion

3.5.1. Impact of Prompt Design and Context Length:

3.5.1.1. Prompting Strategies in Vulnerability Detection:

To evaluate the impact of prompt formulation on the classification performance of Large Language Models (LLMs), two different styles were explored: a question-based prompt (Prompt A) and a completion-based prompt (Prompt B).

Prompt A was designed to guide the model in a direct and explicit way, much like asking a question to a human expert. It mimics a typical interaction where the model is clearly instructed to determine the type of vulnerability in a given code snippet. This prompt encourages the model to focus on answering a specific question based on the context provided.

In contrast, Prompt B was structured to give the impression that the model is completing or continuing a sentence. Rather than explicitly asking for a classification, the prompt leads the model to infer the type of vulnerability as part of a logical continuation. This strategy encourages the model to "think" in a more natural classification setting, without direct instructions.

The comparison between these two prompts aimed to uncover whether the LLM benefits more from an explicitly framed question or from an implicit, sentence-completion formulation. This investigation provides practical insight into how prompt phrasing may affect model behavior in downstream classification tasks related to software security.

Model Variant	Prompt Style	Max length	Accuracy	F1-Macro	F1-Weighted
Baseline	None	512	0.76	0.46	0.73
Prompt A	Question-Based	512	0.75	0.47	0.73

Prompt A	Question-Based	128	0.73	0.36	0.70
Prompt B	Completion-Style	512	0.75	0.46	0.73
Prompt B	Completion-Style	128	0.73	0.36	0.70

Table 17:Comparison of Prompt Styles and Context Lengths on BERT-Based Vulnerability Classification

The results in Table 17 demonstrate that Prompt A provides a slight improvement in macro-level performance, while Prompt B yields similar results to the baseline. Reducing the context length to 128 tokens consistently led to lower performance across all metrics, highlighting the importance of full input context.

3.5.2. Strengths and Limitations of LLM-Based Detection:

Throughout the experiments, BERT-based models were evaluated under multiple configurations, including prompt design variations and input context lengths. These configurations revealed both the strengths and limitations of using large language models (LLMs) for software vulnerability detection.

3.5.2.1. Strenghts:

- **Good Generalization on Well-Represented Classes:** Across all configurations, the models performed consistently well on high-support classes such as "buffer overflow on scanf" and "dereference failure: NULL pointer", achieving F1-scores above 0.75. This shows that LLMs can effectively learn and generalize when sufficient examples are available.
- **Robust Performance Without Prompting:** Surprisingly, the baseline model (without any prompt) achieved the highest accuracy (0.76) and matched the best F1-weighted score (0.73), indicating that BERT is capable of learning complex vulnerability patterns directly from raw code without requiring additional natural language context.
- **Slight Gains in Macro F1 With Prompt A:** Prompt A, designed as a question (“What type of vulnerability is this?”), led to a small improvement in macro F1 (from 0.43 to 0.44), showing potential benefits in helping the model focus more evenly across rare classes.

3.5.2.2. Limitation:

- **Prompt Effect Is Marginal:** Despite expectations, prompt design did not significantly boost performance. Prompt B in particular, which used a completion-style format,

resulted in little to no improvement over the baseline. This suggests that in fine-tuned setups, prompts may not meaningfully influence model decisions.

- **Strong Sensitivity to Input Truncation:** Reducing the input context to 128 tokens consistently harmed performance across all prompts. Both macro F1 and accuracy dropped, indicating that the model needs full code context to detect vulnerabilities reliably. Shorter inputs likely truncate important control or data flow logic.
- **Difficulty with Rare Vulnerabilities:** All configurations struggled with underrepresented classes, such as "arithmetic overflow on floating-point operations", which had F1-scores below 0.25 regardless of prompt usage. This highlights the model's limitation in dealing with class imbalance.
- **Interpretability and Resource Demands:** While not directly measured in this study, LLMs like BERT are computationally expensive and lack interpretability by default. Understanding why a classification was made requires external explainability tools, which limits trust and deployment in security-critical environments.

3.5.3. Scalability, Inference Latency & Resource Consumption:

Large Language Models (LLMs) like BERT and NeoBERT offer strong performance for vulnerability classification tasks. However, this comes at the cost of increased resource demands, training time, and deployment complexity, especially in real-world systems.

- **Scalability:** LLMs are scalable in terms of learning—they can generalize across many vulnerability types without the need for task-specific architectures. However, from an operational standpoint, scalability is constrained by computational limits. For instance, increasing the input context size (e.g., using max_length=512) leads to significantly higher memory usage and longer training time.
- **Inference Latency:** One key observation is that inference latency varies depending on the model architecture and the GPU used. For example, using the NVIDIA A100 GPU, we observed:
  - ✓ BERT completed training in approximately 1.5 hours.
  - ✓ NeoBERT required around 4.5 hours under the same conditions.
  - ✓ This substantial time difference illustrates how architectural complexity impacts inference time—even on high-end hardware.
  - ✓ Reducing max_length to 128 tokens helped improve speed and lower memory usage, but also led to a slight drop in performance, particularly for classes that depend on longer code contexts to be correctly classified.
- **Resource Consumption:**

VRAM Usage: Memory consumption increased with input length and number of classes. Models often required 10–12 GB of GPU memory.

Training Time: Training time varied depending on the model. NeoBERT was consistently slower and heavier than BERT.

3.6. Chapter Summary and Future Directions:

3.6.1. Chapter Summary:

This chapter presented a comprehensive experimental evaluation of various models for software vulnerability detection, with a focus on both traditional machine learning classifiers and large language models (LLMs). We explored the performance of models such as Random Forest, SVM, and XGBoost, as well as pre-trained transformers including BERT, NeoBERT , RoBERTa, and CodeBERT. These models were assessed using a stratified subset of the FormAI dataset, filtered to include only the most frequent vulnerability or error types.

Key experimental dimensions included:

- ✓ Training and validation dynamics, visualized through learning curves,
- ✓ Evaluation on unseen test data, including confusion matrices,
- ✓ Impact of prompt-based input formatting (Prompt A vs Prompt B),
- ✓ Effect of varying context length (max_length) on classification accuracy and model efficiency.

The results revealed that LLMs outperform traditional models in capturing code semantics, especially when augmented with carefully designed prompts. However, prompt design and input truncation (via max_length) significantly influenced detection quality. Prompt A (explicit question-style) generally yielded stronger results than Prompt B (masked-style continuation), though results varied across specific classes.

In addition to raw performance, we analyzed each model's scalability, inference latency, and resource footprint, finding that while BERT-based models offer better accuracy, they incur higher computational costs, particularly on longer sequences or larger batch sizes.

3.6.2. Future Directions:

- ➢ Prompt Optimization: One practical direction to boost model performance without changing

- Deployment Optimization: To make the system feasible for deployment in low-resource environments (e.g., edge devices or servers without GPUs), model optimization techniques such as:

Knowledge Distillation (transferring knowledge from a large model to a smaller one), and **Quantization** (reducing the numerical precision of weights to decrease size and computational load), can be employed. These techniques help reduce inference latency and resource consumption, making LLM-based detection practical for real-time or large-scale scenarios

- Explainability and Developer Interaction: To increase trust and usability, especially for security

analysts and developers, it is important to include explanation mechanisms. Techniques like attention visualization, token attribution, or saliency maps can help users understand why a certain vulnerability label was predicted, leading to more informed debugging and better human-AI collaboration.

General Conclusion

This work has highlighted the remarkable potential of Large Language Models (LLMs) for automating software vulnerability detection. Through a comprehensive and methodologically sound experimental framework, we evaluated multiple transformer-based models including BERT, CodeBERT, and NeoBERT alongside different prompt formulations and input context lengths. The results consistently demonstrated that LLMs significantly outperform traditional machine learning approaches such as Random Forest and SVM, particularly in terms of their ability to understand code semantics and manage multi-class classification tasks.

Several key findings emerged from this study:

Explicit prompts (Prompt A) which directly ask the model a clear question consistently produced better performance than implicit completions (Prompt B).

Reducing input context length from 512 to 128 tokens resulted in a noticeable decline in accuracy and generalization, confirming the importance of rich contextual information for effective code understanding.

Reducing input context length from 512 to 128 tokens resulted in a noticeable decline in accuracy and generalization, confirming the importance of rich contextual information for effective code understanding.

Model performance remains sensitive to class imbalance, especially with underrepresented vulnerability types, highlighting a common challenge in real-world datasets.

Beyond these findings, this thesis opens multiple promising research avenues:

The integration of static analysis features such as control-flow or data-flow insights could help enrich the model's input representation and improve its decision-making.

Automatic prompt engineering using optimization algorithms or reinforcement learning could further enhance performance without the need to modify model architecture.

In conclusion, this research establishes LLM-based detection as a significant step forward toward more intelligent, robust, and explainable software vulnerability analysis systems. With further optimization and real-world integration, such systems have the potential to play a vital role in building more secure and resilient software ecosystems.

Bibliography:

[1] Xu, H., Wang, S., Li, N., Wang, K., Zhao, Y., Chen, K., ... & Wang, H. (2024). Large language models for cyber security: A systematic literature review. *arXiv preprint arXiv:2405.04760*.

[2] Yigit, Y., Ferrag, M. A., Ghanem, M. C., Sarker, I. H., Maglaras, L. A., Chrysoulas, C., ... & Janicke, H. (2025). Generative ai and llms for critical infrastructure protection: evaluation benchmarks, agentic ai, challenges, and opportunities. *Sensors*, *25*(6), 1666.

[3] Zitser, M., Lippmann, R., & Leek, T. (2004, October). Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering* (pp. 97-106).

[4] Fan, J., Li, Y., Wang, S., & Nguyen, T. N. (2020, June). AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th international conference on mining software repositories* (pp. 508-512).

[5] Tamberg, K., & Bahsi, H. (2025). Harnessing large language models for software vulnerability detection: A comprehensive benchmarking study. *IEEE Access*.

[6] Chen, W., Zou, X., Li, G., & Qian, Z. (2020). {KOOBE}: Towards facilitating exploit generation of kernel {Out-Of-Bounds} write vulnerabilities. In *29th USENIX security symposium (USENIX security 20)* (pp. 1093-1110).

[7] Badoux, N., Toffalini, F., Jeon, Y., & Payer, M. (2025). Type++: prohibiting type confusion with inline type information. NDSS.

[8] Noman, H. A., & Abu-Sharkh, O. M. (2023). Code injection attacks in wireless-based Internet of Things (IoT): A comprehensive review and practical implementations. *Sensors*, *23*(13), 6067.

[9] Verma, A. (2023). Insecure Deserialization Detection in Python.

[10] Lee, Y. T., Vijayakumar, H., Qian, Z., & Jaeger, T. (2024). Static detection of filesystem vulnerabilities in android systems. *arXiv preprint arXiv:2407.11279*.

[11] Majdinasab, V., Bishop, M. J., Rasheed, S., Moradidakhel, A., Tahir, A., & Khomh, F. (2024, March). Assessing the Security of GitHub Copilot's Generated Code-A Targeted Replication Study. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 435-444). IEEE.

[12] Piskachev, G., Petrasch, T., Späth, J., & Bodden, E. (2020). AuthCheck: Program-state analysis for access-control vulnerabilities. In *Formal Methods. FM 2019 International Workshops: Porto, Portugal, October 7–11, 2019, Revised Selected Papers, Part II 3* (pp. 557-572). Springer International Publishing.

[13] Abirami, J., Devakunchari, R., & Valliyammai, C. (2015, December). A top web security vulnerability SQL injection attack—Survey. In *2015 Seventh International Conference on Advanced Computing (ICoAC)* (pp. 1-9). IEEE.

[14] Van Der Kouwe, E., Nigade, V., & Giuffrida, C. (2017, April). Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems* (pp. 405-419).

[15] Cheng, X., Wang, H., Hua, J., Xu, G., & Sui, Y. (2021). Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *30*(3), 1-33.

[16] Arusoaie, A., Ciobâca, S., Craciun, V., Gavrilut, D., & Lucanu, D. (2017, September). A comparison of open-source static analysis tools for vulnerability detection in c/c++ code. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)* (pp. 161-168). IEEE.

[17] Jovanovic, N., Kruegel, C., & Kirda, E. (2006, May). Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)* (pp. 6-pp). IEEE.

[18] Zheng, Y., & Zhang, X. (2013, May). Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *2013 35th International Conference on Software Engineering (ICSE)* (pp. 652-661). IEEE.

[19] Ferrag, M. A., Alwahedi, F., Battah, A., Cherif, B., Mechri, A., Tihanyi, N., ... & Debbah, M. (2025). Generative AI in Cybersecurity: A Comprehensive Review of LLM Applications and Vulnerabilities. *Internet of Things and Cyber-Physical Systems*.

[20] Feist, J., Mounier, L., Bardin, S., David, R., & Potet, M. L. (2016, December). Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering* (pp. 1-12).

[21] Farahmandi, F., & Alizadeh, B. (2015). Gröbner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction. *Microprocessors and Microsystems*, *39*(2), 83-96.

[22] Harer, J. A., Kim, L. Y., Russell, R. L., Ozdemir, O., Kosta, L. R., Rangamani, A., ... & Lazovich, T. (2018). Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*.

[23] Zhou, X., Cao, S., Sun, X., & Lo, D. (2024). Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Transactions on Software Engineering and Methodology*.

[24] Harzevili, N. S., Belle, A. B., Wang, J., Wang, S., Ming, Z., & Nagappan, N. (2023). A survey on automated software vulnerability detection using machine learning and deep learning. *arXiv preprint arXiv:2306.11673*.

[25] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., ... & Zhong, Y. (2018). Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*.

[26] Cheng, X., Wang, H., Hua, J., Xu, G., & Sui, Y. (2021). Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *30*(3), 1-33.

[27] Tihanyi, N., Bisztray, T., Ferrag, M. A., Cherif, B., Dubniczky, R. A., Jain, R., & Cordeiro, L. C. (2025). Vulnerability Detection: From Formal Verification to Large Language Models and Hybrid Approaches: A Comprehensive Overview. *arXiv preprint arXiv:2503.10784*.

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

[29] Wang, Y., Wang, W., Joty, S., & Hoi, S. C. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

[30] Dubniczky, R. A., Horvát, K. Z., Bisztray, T., Ferrag, M. A., Cordeiro, L. C., & Tihanyi, N. (2025). Castle: Benchmarking dataset for static code analyzers and llms towards cwe detection. *arXiv preprint arXiv:2503.09433*.

[31] Harzevili, N. S., Belle, A. B., Wang, J., Wang, S., Ming, Z., & Nagappan, N. (2023). A survey on automated software vulnerability detection using machine learning and deep learning. *arXiv preprint arXiv:2306.11673*.

[32] Wartschinski, L., Noller, Y., Vogel, T., Kehrer, T., & Grunske, L. (2022). VUDENC: vulnerability detection with deep learning on a natural codebase for Python. *Information and Software Technology*, *144*, 106809.

[33] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., & Chen, Z. (2021). Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, *19*(4), 2244-2258.

[34] Zou, D., Wang, S., Xu, S., Li, Z., & Jin, H. (2019). $\mu $ µ VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing*, *18*(5), 2224-2236.

[35] Wartschinski, L., Noller, Y., Vogel, T., Kehrer, T., & Grunske, L. (2022). VUDENC: vulnerability detection with deep learning on a natural codebase for Python. *Information and Software Technology*, *144*, 106809

[36] Ferrag, M. A., Alwahedi, F., Battah, A., Cherif, B., Mechri, A., & Tihanyi, N. (2024). Generative ai and large language models for cyber security: All insights you need. *Available at SSRN 4853709*.

[37] Sultan, M. F., Karim, T., Shaon, M. S. H., Wardat, M., & Akter, M. S. (2024). Enhanced LLM-Based Framework for Predicting Null Pointer Dereference in Source Code. *arXiv preprint arXiv:2412.00216*.

[38] Wang, Y., Wang, W., Joty, S., & Hoi, S. C. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

[39] Mahbub, P., & Rahman, M. M. (2024, March). Predicting line-level defects by capturing code contexts with hierarchical transformers. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 308-319). IEEE.

[40] Breton, L. L., Fournier, Q., Mezouar, M. E., & Chandar, S. (2025). NeoBERT: A Next-Generation BERT. *arXiv preprint arXiv:2502.19587*.

[41] Yamagishi, Y., Kikuchi, T., Hanaoka, S., Yoshikawa, T., & Abe, O. (2025). ModernBERT is More Efficient than Conventional BERT for Chest CT Findings Classification in Japanese Radiology Reports. *arXiv preprint arXiv:2503.05060*.

[42] Ye, J., Chen, X., Xu, N., Zu, C., Shao, Z., Liu, S., ... & Huang, X. (2023). A comprehensive capability analysis of gpt-3 and gpt-3.5 series models. *arXiv preprint arXiv:2303.10420*.

[43] Inan, H., Upasani, K., Chi, J., Rungta, R., Iyer, K., Mao, Y., ... & Khabsa, M. (2023). Llama guard: Llm-based input-output safeguard for human-ai conversations. *arXiv preprint arXiv:2312.06674*.

[44] Yang, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., ... & Qiu, Z. (2024). Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

[45] Fan, J., Li, Y., Wang, S., & Nguyen, T. N. (2020, June). AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th international conference on mining software repositories* (pp. 508-512).

[46] Ferrag, M. A., Battah, A., Tihanyi, N., Jain, R., Maimuţ, D., Alwahedi, F., ... & Cordeiro, L. C. (2025). SecureFalcon: Are we there yet in automated software vulnerability detection with LLMs?. *IEEE Transactions on Software Engineering*.

[47] Grahn, D., & Zhang, J. (2021). An analysis of C/C++ datasets for machine learning-assisted software vulnerability detection. In *Proceedings of the Conference on Applied Machine Learning for Information Security, 2021.*

[48] Tihanyi, N., Bisztray, T., Jain, R., Ferrag, M. A., Cordeiro, L. C., & Mavroeidis, V. (2023, December). The formai dataset: Generative ai in software security through the lens of formal verification. In *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering* (pp. 33-43).

[49] Tihanyi, N., Bisztray, T., Ferrag, M. A., Jain, R., & Cordeiro, L. C. (2025). How secure is AI-generated code: a large-scale comparison of large language models. *Empirical Software Engineering*, *30*(2), 1-42.

[50] AI, N. (2024). Artificial intelligence risk management framework: Generative artificial intelligence profile.

[51] Keltek, M., Hu, R., Sani, M. F., & Li, Z. (2024). LSAST--Enhancing Cybersecurity through LLM-supported Static Application Security Testing. *arXiv preprint arXiv:2409.15735*.

[52] Tihanyi, N., Jain, R., Charalambous, Y., Ferrag, M. A., Sun, Y., & Cordeiro, L. C. (2023). A new era in software security: Towards self-healing software via large language models and formal verification. *arXiv preprint arXiv:2305.14752*.

[53] Zhang, L., Zou, Q., Singhal, A., Sun, X., & Liu, P. (2024, June). Evaluating Large Language Models for Real-World Vulnerability Repair in C/C++ Code. In *Proceedings of the 10th ACM International Workshop on Security and Privacy Analytics* (pp. 49-58).

[54] Shestov, A., Levichev, R., Mussabayev, R., Maslov, E., Cheshkov, A., & Zadorozhny, P. Finetuning Large Language Models for Vulnerability Detection. arXiv 2024. *arXiv preprint arXiv:2401.17010*.

[55] Tihanyi, N., Bisztray, T., Ferrag, M. A., Jain, R., & Cordeiro, L. C. (2024). Do neutral prompts produce insecure code? formai-v2 dataset: Labelling vulnerabilities in code generated by large language models. *arXiv preprint arXiv:2404.18353*.