UNIVERSITY OF CATANIA

DOCTORAL THESIS

AI-Driven Software Security: From Static Vulnerability Detection to Adversarially Robust Defense

Author:
Claudio CURTO

Supervisor:
Prof. Daniela GIORDANO
Ing. Daniel Gustav INDELICATO

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Department of Electrical Electronic and Computer Engineering (DIEEI)

October 20, 2025

"The best worst mistake I could ever make. And I would make it again."

UNIVERSITY OF CATANIA

Abstract

Department of Electrical Electronic and Computer Engineering (DIEEI)

Doctor of Philosophy

AI-Driven Software Security: From Static Vulnerability Detection to Adversarially Robust Defense

by Claudio CURTO

The increasing complexity of modern software systems, coupled with the continuous rise in reported vulnerabilities, has underscored the urgent need for scalable and automated approaches to vulnerability detection. In recent years, transformer-based architectures have emerged as state-of-the-art solutions for Static Application Security Testing (SAST), consistently outperforming traditional rule-based methods. However, despite these advances, significant challenges remain concerning the optimal formulation of the detection task, the quality and diversity of available datasets, and the robustness of current detectors against adversarial manipulations.

This thesis addresses these three complementary challenges through a set of interrelated studies. First, we investigate different architectural choices for vulnerability detection, starting with an analysis of multitask learning and subsequently evaluating the effectiveness of Llama 3 and CodeLlama for SAST. Our results show that these models achieve competitive performance in both vulnerability detection and line-level localization, with CodeLlama benefiting from its specialized pre-training on source code. Second, we conduct a systematic evaluation of existing vulnerability datasets, analyzing their biases, limitations, and impact on model generalization. We find that data quality and diversity are more decisive than model complexity in determining real-world applicability. Strategies such as dataset aggregation and duplication removal prove effective in mitigating bias and improving generalization across projects and vulnerability types. Third, we assess the robustness of state-of-the-art detectors under adversarial scenarios. By applying semantics-preserving code transformations—such as identifier renaming and dead code injection—we demonstrate that current transformer-based models are highly sensitive to lexical and semantic perturbations, exposing critical weaknesses in security-critical contexts.

Together, these contributions provide a comprehensive perspective on the strengths and limitations of modern AI-based vulnerability detectors. By bridging architectural innovation, dataset reliability, and robustness analysis, this thesis advances the state of automated software vulnerability detection and outlines pathways toward more generalizable, trustworthy, and resilient solutions.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Prof. Daniela Giordano, for her unwavering support, guidance, and encouragement throughout my PhD journey. Her expertise and human support have been invaluable in shaping both my research and personal growth.

I am also profoundly thankful to Ing. Daniel Gustav Indelicato for his insightful advice and constructive feedback, which have significantly contributed to the quality of my work. I also wish to thank EtnaHitech and Darwin Technologies for their support and financial assistance during my studies.

I am grateful to all my colleagues at the Perceive Lab for creating an inspiring and friendly research environment. Special thanks go to Amelia, Miriana, Simone, Salvo, Sarah, and Mariaelena for their help and support during the most challenging times of my PhD, and for always inspiring me to give my best.

I would also like to express my appreciation to Prof. Andrea Continella, Prof. Luca Mariot, Prof. Thijs van Ede, and the Semantics, Cybersecurity and Services (SCS) Group at the University of Twente, for welcoming me during my research stay, for their collaboration and fruitful discussions.

Most of all, I want to thank my family for their unconditional love and support.

Thanks to Manlio, Pietro, and Alex for being the best friends I could ever ask for.

Thanks to Federico, Giorgio, Manuele, Filippo, and Alfonso for being my piece of Italy in the Netherlands and for all the unforgettable moments we shared together.

And finally, thank you Giorgia, my heart and my joy, for being there at my highest and lowest — always my anchor and my wings when I needed them the most.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, the application of machine learning techniques to software vulnerability detection has received increasing attention, driven by the growing scale and complexity of software systems and the corresponding rise in security threats. Vulnerabilities in source code represent a major risk, as they can lead to data breaches, service interruptions, and compromises of system integrity. The steady growth in the number of reported Common Vulnerabilities and Exposures (CVEs) [36] — as shown in Figure 1.1 — highlights the persistence of this problem and the urgent need for scalable and automated detection approaches.

Static Application Security Testing (SAST) remains one of the most widely adopted strategies for identifying vulnerabilities early in the development cycle. Since programming languages exhibit both syntactic and sequential characteristics, many SAST techniques have been reformulated as Natural Language Processing (NLP) tasks. This shift has enabled the adoption of transformer-based architectures [46], which currently represent the state of the art in vulnerability detection. Pre-trained models such as CodeBERT [15], GraphCodeBERT [20], and related architectures leverage large-scale code corpora to capture both syntactic and semantic representations, consistently outperforming traditional rule-based techniques [34, 16, 10].

Despite these advances, several fundamental challenges remain unresolved. A first challenge relates to the effectiveness of model architectures. While transformer-based detectors have shown remarkable performance in controlled benchmarks, questions remain regarding the optimal formulation of the detection task itself. Approaches such as binary classification, multiclass categorization, and multitask learning offer different trade-offs between accuracy, granularity, and interpretability. Understanding these trade-offs is crucial for designing detectors that can support security analysts not only in flagging vulnerable code but also in providing contextual

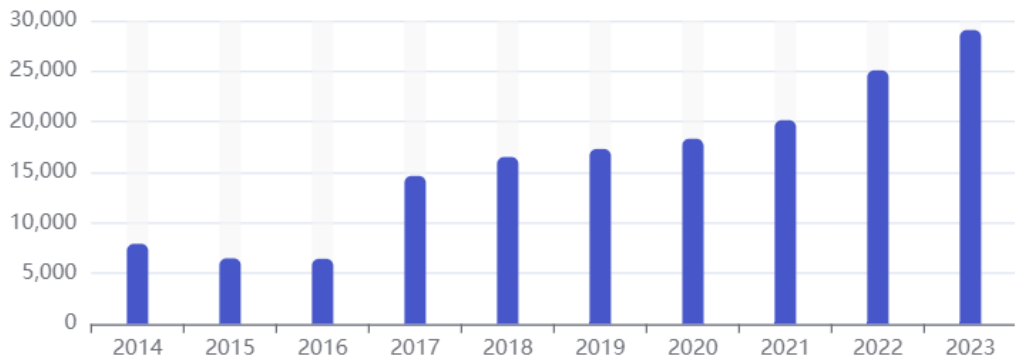


FIGURE 1.1: CVEs' publishing trend by year until 2023.

information about the type and nature of the vulnerability.

A second issue concerns the quality of datasets used for training and evaluation. Public vulnerability datasets often present duplication, noisy or incomplete labels, and limited coverage of real-world software, leading to over-optimistic results and poor generalization when models are deployed on unseen projects or vulnerability types. Improving dataset diversity and reliability has therefore become a key requirement for developing trustworthy detectors.

Finally, even state-of-the-art detectors suffer from limited robustness against adversarial manipulations. Unlike random noise, adversarial attacks in software vulnerability domain consist of semantics-preserving transformations of source code —- such as identifier renaming, reordering of independent statements, or injection of dead code -— crafted to mislead the model while preserving the original functionality. These perturbations exploit weaknesses in the learned representations of the models, and recent studies have shown that transformer-based architectures are especially sensitive to such changes. This poses a serious risk in security-critical contexts, where undetected flaws (false negatives) or excessive false alarms (false positives) can have severe implications.

This thesis addresses these three complementary challenges by presenting the following research contributions organized into three distinct yet interconnected research themes:

- **Model's architectures for vulnerability Detection**: an evaluation of transformer-based and multitask approaches, comparing them to state-of-the-art baselines and analyzing their effectiveness at function-level and line-level localization.
- **Dataset quality and generalization**: an in-depth analysis of existing vulnerability datasets, their limitations, and strategies for improving coverage and representativeness.
- **Adversarial robustness of vulnerabilities detectors**: A systematic Study of Semantics Preserving adversarial Attacks Against Transformer-Based Models, with a Focus on Understanding Their weaknesses and Outlining Directions for Building More Resilient Systems

Together, these contributions provide a comprehensive overview of the strengths and weaknesses of current AI-based vulnerability detection approaches, highlighting the interplay between data quality, model design, and robustness. By integrating these perspectives, this thesis aims to advance the state of automated software security analysis toward more reliable, generalizable, and robust solutions.

1.1 Thesis Organization

This thesis is organized into eight main chapters (the first one is this introduction).

Chapter 2 – Background introduces the fundamental concepts and methodologies underpinning this work. It reviews the principles of software vulnerability detection, static application security testing (SAST), and recent advances in machine learning and natural language processing for source code analysis. This chapter also provides the necessary theoretical and technical foundations to contextualize the contributions presented in the subsequent chapters.

Chapter 3 – MultiVD: a Transformer-based Multitask Approach for Software Vulnerability Detection presents a multitask learning framework designed to simultaneously address vulnerability detection and categorization. The chapter discusses the motivation for adopting a multitask approach over traditional multiclass and binary formulations, details the experimental evaluation, and highlights how multitask models can provide richer and more informative outputs for security analysts.

Chapter 4 – Can a Llama be a Watchdog? Exploring Llama 3 and Code Llama for Static Application Security Testing investigates the applicability of large language models, specifically Llama 3 and Code Llama, to vulnerability detection tasks. The chapter compares their performance against state-of-the-art baselines, evaluates their ability to localize vulnerabilities at the line level, and explores how preprocessing techniques such as comment cleaning can improve detection accuracy without degrading other downstream tasks.

Chapter 5 – Addressing the C/C++ Vulnerability Datasets Limitation: the Good, the Bad and the Ugly focuses on the role of dataset quality and diversity in the development of reliable vulnerability detectors. It examines existing datasets, their biases and limitations, and evaluates the generalization ability of models across unseen projects and vulnerability types. The chapter also discusses how combining datasets and removing duplicates can improve robustness, while emphasizing the need for more representative data to bridge the gap with real-world scenarios.

Chapter 6 – Adversarial Attacks to AI-based Vulnerability Detectors analyzes the robustness of state-of-the-art models under adversarial conditions. It investigates different attack strategies based on semantics-preserving code transformations, such as identifier renaming and dead code injection, and quantifies their impact on various transformer-based detectors. The results underline the vulnerabilities of current approaches and provide insights into the architectural factors influencing resilience.

Chapter 7 – Future Works outlines potential directions for extending the contributions of this thesis. It discusses promising avenues such as improving dataset curation, exploring hybrid architectures combining transformers and graph-based models, developing defense mechanisms against adversarial perturbations, and leveraging large language models for more explainable and interactive vulnerability detection.

Chapter 8 – Conclusions summarizes the main findings of this thesis, highlighting how the conducted studies contribute to a deeper understanding of the interplay between data quality, model design, and robustness in AI-based vulnerability detection. It also reflects on the broader implications of this work for advancing secure software development practices.

Chapter 2

Background

2.1 Software Vulnerability Assessment

Ghaffarian et al. [19] gave this definition for software vulnerability:

A software vulnerability is an instance of a flaw, caused by a mistake in the design, development, or configuration of software such that it can be exploited to violate some explicit or implicit security policy.

A software security test consists in a validation process of a system or application with respect to certain criteria. There are several approaches to test the vulnerability of a software; these approaches can be categorized in three groups:

- Static analysis: the vulnerability assessment is performed directly on the application's source code, usually before its deployment. This form of testing can be performed using pattern recognition techniques or, in a machine learning scenario, by NLP-based techniques (LSTM, GNN, transformers). This typology of approach has the advantage of detecting, in the best case, all the vulnerabilities in the code, at the cost of a high false positives rates, that may be caused from several factors as limited context awareness, complexity of the code or the use of custom libraries and frameworks.
- Dynamic analysis: the assessment is performed on the running application, monitoring its behaviour. Generally called "penetration testing", the focus of the test is to validate the application behaviour following some specific inputs given by the user. Potentially, however, there are cases where it is possible to have nearly infinite different inputs, so it would be impossible to test every scenario (some vulnerability can be lost).
- Hybrid analysis: a hybrid approach takes advantage of strengths and limitations of static and dynamic approaches. It is possible to have a static approach that leverages some dynamic properties to individuate false vulnerabilities and so on.

Ghaffarian et al. [19] identified four classes of methodologies applied in vulnerability assessment:

- Anomaly detection approaches
- Vulnerable code pattern recognition
- Software metrics based approaches
- Miscellaneous approaches

Vulnerable code pattern recognition is the one explored in this Thesis. In this method the search of vulnerability patterns is performed directly on the program source code, without the requirement of running the application. It is a very promising approach in the field of vulnerability discovery, but characterized by the subsequent aspects that must be addressed to produce effective vulnerability detection tools:

1. Vulnerability discovery systems must be able to identify the type of vulnerability. It is a very important aspect for the security analysts, for example to assign priority to the tasks. Furthermore, a crucial aspect is the number of detectable vulnerabilities, as a higher value involves a more powerful vulnerability detection system.

2. Precise vulnerability localization in the code. It is related to the granularity of the approach, usually coarse (at function or project level). Knowing the exact location of the detected vulnerability may have a significant impact on its subsequent correction.

3. For an effective modeling and discovery of vulnerabilities, some information about different aspects of the analyzed software is needed, like syntactic information, control flow information, and data flow information. It is suggested that using deep-learning methods will produce very powerful vulnerable code pattern recognition systems, representing a very promising research area. Transformer-based language models applied to programming language have achieved state-of-the-art performances, demonstrating their high potential in this field [15].

2.1.1 Common Vulnerability and Exposures (CVE)

The Common Vulnerabilities and Exposures (CVE) system is a publicly accessible repository of documented security vulnerabilities and exposures in software and hardware, maintained by the MITRE Corporation. Each entry is assigned a unique identifier, known as a CVE ID (e.g., CVE-YYYY-NNNNN), which provides a standardized reference for reporting and discussing specific flaws. This approach enables consistent communication among security professionals, vendors, and the wider community, and has become a cornerstone of modern vulnerability management practices.

The CVE system offers several advantages. First, the use of standardized identifiers facilitates interoperability and data exchange across different security tools and databases, fostering a common language within the cybersecurity domain. Second, the CVE list is openly available, ensuring that information about known vulnerabilities can be accessed and leveraged by organizations worldwide. Finally, CVEs support effective vulnerability management by allowing organizations to: (i) prioritize remediation efforts on the most critical threats, (ii) track vulnerabilities relevant to their systems, and (iii) remediate them by applying available patches or mitigations.

The initiative is administered by the MITRE Corporation[1], a non-profit organization funded in part by the U.S. Department of Homeland Security, which ensures the continuous maintenance and evolution of the CVE system.

[1] https://www.mitre.org/

2.1.2 Common Weaknesses Enumeration (CWE)

The Common Weakness Enumeration (CWE) is a community-driven, MITRE-maintained taxonomy that provides a standardized language for describing and categorizing common software and hardware weaknesses. Unlike the CVE system, which documents specific, real-world vulnerabilities, CWE focuses on the underlying design, implementation, or configuration flaws, such as buffer overflows or cross-site scripting, that may lead to security vulnerabilities. Its primary goal is to support education, prevention, and improved awareness of recurring weaknesses, enabling organizations and developers to address security risks at their root cause. Key aspects of CWE include:

- **Categorization:** Weaknesses are organized into a hierarchical taxonomy, offering a common framework for describing and classifying flaws.
- **Root Cause Focus:** CWE highlights the fundamental design or coding issues that introduce vulnerabilities, rather than documenting individual incidents.
- **Community-Driven Development:** The list is developed and maintained by MITRE in collaboration with industry, academia, and government organizations.

Examples of common weaknesses include:

- **CWE-79:** Cross-Site Scripting (XSS) – Improper neutralization of input during web page generation.
- **CWE-89:** SQL Injection – Improper neutralization of special elements used in SQL commands.
- **CWE-22:** Path Traversal – Improper limitation of a pathname to a restricted directory.
- **CWE-119:** Buffer Overflow – Improper restriction of operations within the bounds of a memory buffer.

It is essential to distinguish CWE from CVE. While CWE provides a catalog of generic weakness types and their root causes (e.g., XSS or buffer overflow), CVE identifies specific instances of such weaknesses exploited in real-world software and systems. In other words, CWE describes *what kinds of weaknesses exist*, while CVE documents *where they occur*.

2.2 Deep Learning Architectures for Vulnerability Detection

Recent advances in machine learning for software security have led to the development of a wide range of models aimed at detecting vulnerabilities in source code. These models can be broadly categorized into two main families: sequence-based models and graph-based models.

2.2.1 Sequence-based

Sequence-based models treat source code as a linear sequence of tokens or subtokens, often leveraging architectures originally designed for natural language processing [25]. Notable examples include models based on LSTMs, CNNs, and more

recently, transformers such as CodeBERT [15] or GraphCodeBERT [20]. These models typically rely on pretraining on large corpora of code and are fine-tuned for downstream tasks such as vulnerability classification [9]. Transformers addressed the limitations associated with RNN (Recurrent Neural Network) architectures, through their attention mechanism and parallel processing, reaching higher levels of efficiency and performance, especially when handling complex linguistic structures. These capabilities extend to code analysis and related tasks as code summarization, clone detection and, of course, vulnerability detection. However, a major limitation of these models lies in their inability to fully exploit the structured and semantic nature of programming languages. Since they treat code as flat text, they are often sensitive to superficial modifications—such as renaming variables, reordering independent statements, or inserting dead code—that do not alter program behavior but can affect the model's prediction. As a result, sequence-based models can be vulnerable to semantics-preserving adversarial attacks, which manipulate the form of the input without affecting its meaning.

Transformers and Self-Attention

Transformers-based architectures have gained immense popularity due to their ability to capture long-range dependencies and context in sequential data, making them highly effective especially in Natural Language Processing (NLP) tasks [27]. The transformer architecture generally consists of two main components: the encoder and the decoder. The encoder is responsible for analyzing the input sequence, while the decoder generates the output sequence based on that encoded representation. In this study, we adopt an encoder-only architecture, which is more suitable for classification tasks.

A key element of the transformer architecture is the attention function, which can be described as mapping a query and a set of key-value pairs to an output. Attention is computed as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

where Q, K, V are respectively the Query, Key, Value vectors and d_k is the dimension of Q and K, used as a scaling factor. In practice, the attention function is computed on a set of Queries simultaneously, packed together in a matrix, as the Keys and the Values. This particular attention is also called "Scaled-Dot-Product Attention", since it is computed by the dot product of the queries with the keys, divided each by $\sqrt{d_k}$ and applying a softmax function to obtain the weights on the values [46]. The attention process is performed in parallel on h linearly projected queries, keys and values on different linear projections to d_k, d_k, d_v dimension, respectively. The results are then concatenated and projected one last time. This is called multi-head attention and allows the model to jointly attend to information from different representation subspaces at different positions.

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O$$

where $head_i = Attention(QW_i^O, KW_i^K, VW_i^V)$.
Self-attention is an attention mechanism that relates different positions within a single sequence in order to compute a context-aware representation of that sequence. Self-attention is one way the transformer uses multi-head attention. In the self-attention layers – contained inside the encoder – all of the key, values and queries

come from the same place, that is the output of the previous layer in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder. Self-attention is also used by the decoder. It allows each position in the decoder to attend to all positions in the decoder up to and including that position.

BERT and BERT-based Architectures

Bidirectional Encoder Representation from Transformers (BERT) has been introduced by Devlin et al. in 2018 [13] and has achieved remarkable results across a wide range of NLP task (e.g., question answering, sentiment analysis, text classification and more). Different variants of BERT have been released, including models for specific domains or models with improved performances like CodeBERT [15] to handle programming languages, understanding source-code data and programming related tasks. It is pre-trained with a 20 GB source code corpus with bimodal instances of NL-PL (Natural Language - Programming Language) pairs, unimodal codes (not paired with natural language texts) and natural language texts without paired codes (2.1M bimodal datapoints and 6.4M unimodal codes across the programming languages Python, Java, JavaScript, PHP, Ruby, Go). CodeBERT has been pre-trained on two tasks: Masked Language Modelling (MLM) and Replaced Token Detection (RTD).

Sequence-based Models for Vulnerability Detection

Zou et al. [54] presented *µVulDeePecker*, the first deep learning-based system for multiclass vulnerability detection. The underlying system architecture is constructed from Bidirectional Long-Short Time Memory (BLSTM) networks and aims to fuse different kinds of features from pieces of code (called code gadgets) and code attention to accommodate different kinds of information. For this purpose, the authors created from scratch a dataset and used it to evaluate the effectiveness of the model. The dataset contains 181,641 code gadgets, with 43,119 vulnerable units and 40 types of vulnerability in total. The authors also tested *µVulDeePecker* on 10 versions of 3 real-world software products and manually examined them to overcome the lack of ground truth. The model achieved high performance both when tested on their own test set (94.22% F1) and on real world software (94.69% F1). The similar results are justified by the reduced number of data and classes in the real-world software (only 16 classes versus the 40 in the synthetic test set), underlying a possible limitation once tested on more complete and various real-world datasets. Furthermore, *µVulDeePecker* can detect vulnerability types, but cannot pin down the precise location of a vulnerability in the code.

Fu et al. [16] introduced LineVul, a transformer-based vulnerability prediction system with high performance both on function-level detection and line-level detection tasks. LineVul strengths are related to its architecture. The model can be seen as a composition of three components: CodeBERT [15], Byte Pair Encoding (BPE) Tokenizer and a single linear layer classifier. BPE is a data compression algorithm, very popular in NLP tasks for its high efficiency in building small vocabularies for text tokenization. The strength of the algorithm is its ability to handle rare or out-of-vocabulary words, using a subword tokenization approach and leading to a better generalization and coverage, especially in cases where the training data may have limited vocabulary coverage. As demonstrated in the paper, the combined use of the pretrained CodeBERT model and Byte Pair Encoding is the key for the high performances of the model: there is a 50% F1-score reduction when using a word-level

tokenization and a 11% reduction when using non-pretrained weights to initialize BERT.

Mamede et al. [34] explored different BERT-based models' performances (CodeBERT [15] and JavaBERT [11]) for multi-label classification of Java vulnerabilities, training them on the Juliet synthetic dataset[2]. The studied models showed high performance when tested on synthetic data and good generalizability when tested on unknown vulnerabilities, related with the kind of vulnerabilities which the models have been trained on, leveraging their belonging Software Fault Patterns (SFP). However, it is pointed out that using only synthetic data is insufficient, since the models' performance severely degrade when facing real-world scenarios. Indeed, all the studied models showed a great loss in performance when tested with real-world data, with a 50% and 58% reduction in F1-score and recall respectively for JavaBERT, indicating that the model can recognise vulnerable patterns but stumble in selecting the type of vulnerability (CWE). Generally, it has been observed that all the model suffer of an high false negative rate when tested in realistic contexts.

2.2.2 Graph-based

In graph-based models programs are represented as graphs in which nodes correspond to program entities (e.g., variables, expressions, control constructs), and edges capture relationships such as syntax (Abstract Syntax Trees, AST), control flow (Control Flow Graph, CFG), and data flow (Data Flow Graph, DFG). These views can be combined into a unified representation known as the Code Property Graph (CPG). Graph-based approaches typically leverage Graph Neural Networks (GNNs) – such as GCNs, GATs, or GGNNs – to process these structured representations. GNNs allow for message passing between nodes, enabling the model to reason over long-range dependencies and semantic interactions within the code. This makes them particularly suitable for tasks like vulnerability detection, where understanding control and data dependencies is crucial. Recent works have also proposed hybrid architectures that combine transformer-based models with graph-based learning, by using pre-trained transformer embeddings as input features to GNNs [31]. These GNN+Transformer models aim to benefit from both the contextual understanding of language models and the structural reasoning capabilities of graphs.

Graph-based Models for Vulnerability Detection

Li et al.[29] propose a fine-grained interpretable model that constructs heterogeneous graphs by integrating multiple code views, including ASTs and Control/-Data Flow dependencies. Their method applies a GNN to propagate information across these different structures, while introducing attention-based mechanisms to highlight the most security-relevant code regions. This not only improves detection accuracy but also enhances interpretability, allowing practitioners to trace back the model's predictions to specific code elements, thereby making the results more transparent and actionable for security auditing.

Tian et al.[44], on the other hand, proposed an approach for vulnerability detection focusing on security-critical flaws such as buffer overflows, null pointer dereference, and memory corruption, addressing the challenge of representing hierarchical syntactic information by decomposing ASTs into fine-grained substructures. Instead

[2]https://samate.nist.gov/SARD/test-suites/111

of treating the AST as a monolithic tree, their approach leverages AST decomposition to capture local context (e.g. statement-level dependencies) while preserving global semantic relationships through graph learning. By combining these decomposed structures with GNN-based reasoning, their framework achieves improved robustness against distribution shifts and outperforms prior methods on multiple benchmark datasets. The evaluation was carried out on widely used datasets such as Devign and DiverseVul, with particular attention to generalization scenarios, including unseen projects and previously unobserved vulnerability types.

Chapter 3

MultiVD: a Transformer-based Multitask Approach for Software Vulnerability Detection

3.1 Introduction

Recentely, transformer-based approaches emerged as state-of-the-art in vulnerability prediction tasks, both on function-level and line-level vulnerability prediction [16], [21]. However, there are various open challenges in this field, as the classification of the predicted vulnerability and the model's generalization capability over different projects [26], [6].

To date, multiclass vulnerability classification has not been fully explored, due to the lack of adequately varied and quality datasets. In 2021 Zou et al. [54] proposed the first multiclass vulnerability classifier, namely $\mu VulDeePecker$. However, their system suffers of two limitations: it is unable to locate the vulnerability in the code and it is trained on a synthetic dataset. The datasets nature is a crucial aspect to consider, as a model trained on a synthetic dataset will be limited to detecting only the simple patterns present in the data, which seldom occur in real life [4]. In contrast, real-world datasets are derived from real-world sources and generally only contain functions that went through vulnerability-fix commits. In this case the model will be able to learn real and more complex patterns.

In this study we explore two possible classification approaches: multiclass and multitask classification. We do this first by fine tuning the CodeBERT[15] architecture to perform a multiclass classification of the known vulnerabilities, to explore the model vulnerability pattern recognition capabilities; second, by improving the original approach with the addition of two classification heads to the model, performing simultaneously a binary classification task for vulnerability detection and a multiclass classification task for vulnerability categorization. The idea is to leverage the multitask approach to provide a more informative result from the model, while, at the same time, enhance the model performance both on the detection and classification fields. Finally, we compare the attention-based line-level detection performance between LineVul and our multitask model to show the effects of the new approach to the original task. In the last step of our study, we explore if it is possible to improve the final results of the multitask model by a different handling of the loss functions.

3.2 Background

3.2.1 Multitask Learning

Multitask learning is a promising area in machine learning that aims to leverage useful information contained in multiple learning tasks to help learn a more accurate learner for each task [51]. The idea of multitask learning is to train the same model to resolve two or more different tasks at the same time, improving its performance on one individual task leveraging shared representations and learning from each other's data. In the case of vulnerability classification problems, common security tools adopt a binary classification approach (given a function, they tell if is vulnerable or not), but this approach is not informative enough for a security analyst, since it does not give a satisfying amount of information about that specific vulnerability. In this study, we describe the problem of software vulnerability detection as a combination of two classification tasks: a multiclass classification of CWE (Common Weakness Enumeration) and a binary classification (whether the code is vulnerable or not). In the next section we describe how this approach is implemented.

3.3 Methodology

3.3.1 Classification

The adopted model architecture extends the structure defined by Fu et al.[16], consisting of three components: BPE tokenizer, CodeBERT and the linear classifier. The linear classifier has been updated for the multiclass and the multitask implementation. The decision to adopt a multitask approach, rather than a traditional multiclass setup with an additional "non-vulnerable" class, stems from a careful analysis of the BigVul dataset construction and data gathering methodology, as described in Subsection 3.3.3. The underlying idea is to leverage the potential informational content related to a specific CWE that may be present even within a function labeled as non-vulnerable, yet still connected to that vulnerability through its commit history. By framing the problem as a multitask task, it becomes possible to exploit this subtle relationship, allowing the model to learn richer representations that capture both vulnerability presence and CWE-specific characteristics.

For the multiclass approach it consists of a single linear layer with 15 output neurons. For the multitask approach the classifier is built with two classification heads, one for the multiclass classification with 15 output neurons (one for each CWE to classify) and the other for the binary classification (vulnerability target). For both tasks, we adopt Cross-entropy as loss function. When training the multitask model, a weighted loss is computed as

$$loss_W = \alpha * loss_M + \beta * loss_B$$

where $loss_M$ is the multiclass loss, $loss_B$ is the binary loss, α and β are two arbitrary parameters that assume values between 0 and 1, with 0 excluded.

3.3.2 Localization

For line-level vulnerability localization we use the approach by Fu et al. [16], leveraging the attention scores assigned to the tokens by the model. The idea of this approach is that high attention tokens are likely to be vulnerable tokens, so lines

with higher attention score are the ones with higher probability to contain the vulnerability. This is done by obtaining the self-attention scores from the trained model for every sub-word token and then integrate those scores into line scores. Specifically, a whole function is split to in lists of tokens where every list represent a line (the split is done by the newline control character \n). Every token in a list will have an associated cumulative attention score, so, for each list of token scores, we summarize it into one attention line score and rank all the line scores. The ranking of the lines based on the relative attention scores will show the lines with higher probabilities to be the location of the vulnerability. This approach is applied considering the true positive outputs of the vulnerability detection task, so in our case we take into account binary predictions only.

3.3.3 Dataset

For a fair comparison with LineVul we use the same benchmark dataset, provided by Fan et al. [14], namely BigVul. BigVul dataset is the only vulnerability dataset that provides line-level ground-truth, necessary to our study to compare line-level prediction performance between the two models. In this section, we provide a brief overview of the dataset, while a detailed description of BigVul and all the studied datasets can be found in Chapter 5.

All the functions in the dataset are assigned a CWE ID describing a vulnerability type, even the not vulnerable functions. This is explained analyzing the way the functions are gathered. The data collection procedure is structured in three steps.

1. First, Fan et al. perform a scraping procedure of the CVE database, collecting all the vulnerabilities information until 2019.

2. From all the entries collected, the ones with a github reference link pointing to a code repository to retrieve the vulnerable projects are selected. For every project, the commit history is extracted.

3. Each retrieved commit from the history is considered as a mini version of its project, and every mini version is mapped to the relative CVE information retrieved in the first step. For each of the commits explicitly considered as relevant, they extracted the code changes that fixed the vulnerability; in this way, it was possible to build the vulnerable code. All the other not relevant commits were considered as not vulnerable.

So, different functions are gathered from the same reference link, reporting both vulnerable and not vulnerable samples, all sharing the same CVE information previously gathered. These functions, even if not vulnerable, may share useful information to recognize some hidden pattern typical of that CWE. From this perspective, we want to explore the results of a classification-based approach considering both vulnerable and not vulnerable samples for CWE classification.

To avoid possible inconsistencies between the data, we obtained the dataset from the LineVul GitHub repository[1]. The dataset in the repository is available both in its split version (train, validation and test split) and unsplit. We obtained the unsplit one, that came with a total of 4.841.688 samples. However all these samples can't be used as they present inconsistent entries e.g. None/NaN values, missing CVE labels or CWEs as pieces of string or code instead the ID. We performed a cleaning

[1] https://github.com/awsm-research/LineVul/tree/main/data

procedure removing all the samples where the CWE ID is not in the format "CWE-number", reducing their number to 146,625 functions, with 7,117 vulnerable functions. Another important aspect of the dataset is its high imbalance among all the different CWE IDs and between vulnerable and not vulnerable functions. This is strictly related to the fact that some vulnerabilities may occur less frequently than others. To get more reliable results, we restricted the multiclass classification to the top-15 CWE classes in terms of distribution in the dataset. The selected classes are reported in Figure 3.1. In the end, the final unbalanced dataset used is composed by 126,313 functions, with 7,089 vulnerable ones. Considering the high imbalance between vulnerable and not vulnerable samples, we test the model performance by training it with two variations of the dataset: unbalanced and balanced, the latter obtained by undersampling the not vulnerable occurrences. The resulting balanced dataset is composed by 14,178 functions equally distributed between vulnerable and not vulnerable. For all tests, the datasets are split into train, validation, and test sets with 80/10/10 ratio, with validation and test sets stratified to keep the same CWE ID class distribution as the train set.

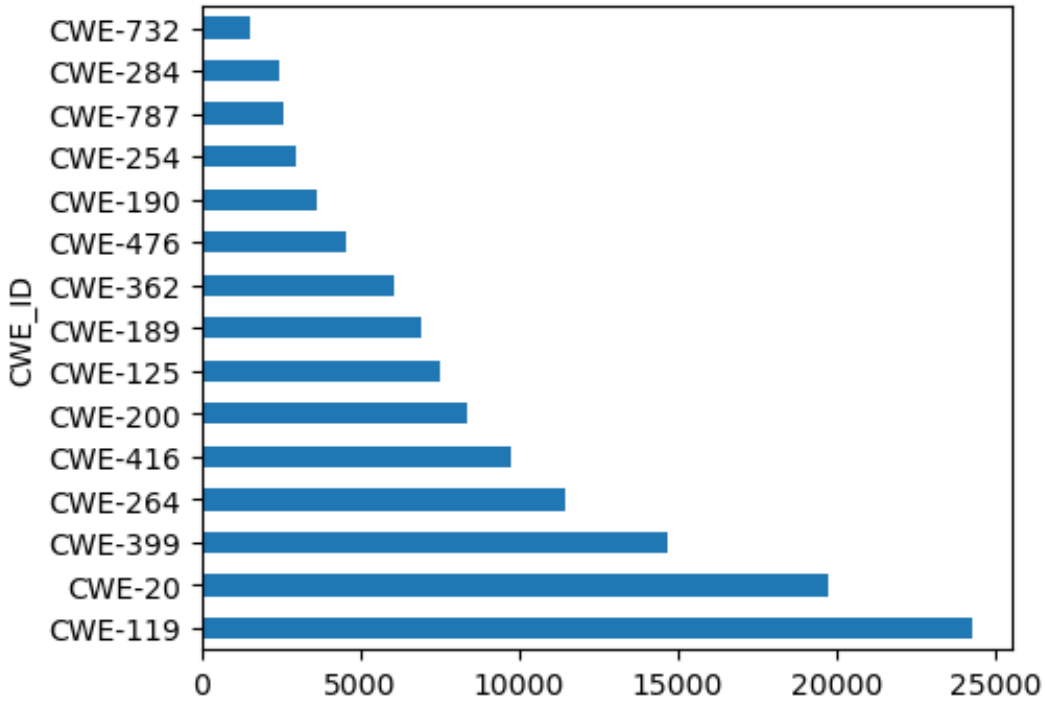


FIGURE 3.1: Top 15 CWEs in the dataset.

3.3.4 Evaluation Metrics

Accuracy, precision, recall and F1-score are computed for training, validation, and testing phases. We take note of the best F1-score on the validation set to save the model checkpoint for testing. For the line-level vulnerability detection task, we evaluate the model using two metrics: Top-10 Accuracy and Initial False Alarm (IFA). Top-10 Accuracy measures the percentage of functions where at least one truly vulnerable line appears within the top 10 ranked lines, based on their cumulative attention scores. The rationale is that, in practice, security analysts are unlikely to consider recommendations beyond the top-10 positions. This metric is computed

by checking whether any ground-truth vulnerable line index falls within the top-10 ranking for each function. If so, the function is counted as correctly localized. Initial False Alarm (IFA) quantifies the number of non-vulnerable lines that an analyst would need to inspect before reaching the first actual vulnerable line within the ranking. Lower IFA values are preferred, since they indicate fewer wasted inspections and thus greater efficiency for the analyst. Formally, given the previously computed ranking of lines, the IFA is determined by identifying the position of each vulnerable line and then taking the minimum among them (i.e., the first appearance of a vulnerable line). For instance, if the first vulnerable line occurs in the 5th position, the analyst would have to inspect four non-vulnerable lines before reaching it, yielding an IFA of 4.

3.4 Experiments Setting

For the model implementation, the pre-trained CodeBERT tokenizer and CodeBERT are downloaded from the HuggingFace repository. For the multiclass approach the linear classifier consists of a single linear layer with 15 output neurons; for the multitask approach the classifier is implemented to return a couple of outputs, one for multiclass classification and the other for binary classification. The training is performed on a NVIDIA RTX A6000 GPU. For the hyperparameter aspect, we leave the CodeBERT default settings unchanged and choose $2 * 10^{-5}$ for the learning rate, with AdamW as the optimizer. The model is trained for 10 epochs with cross-entropy as loss function for all the classifications. The multitask final loss is first computed as an average of the two, while in a second experiment we explore a different approach with a weighted loss.

3.5 Research Questions

During the study we examine the following research questions:

1. Which approach (multiclass vs multitask with our modified architecture) demonstrates the highest performance?
2. What are the effects of the multitask approach in vulnerability localization?
3. Does a weighted loss improve the model's performance compared to using an average loss?

RQ1. Which approach demonstrates the highest performance?

During the training of the model, we take note of the test metrics for each task (multiclass, binary and multitask) and for two versions of the dataset, i.e. unbalanced and balanced, plus an additional test with only vulnerable sample in the case of the multiclass classification. Finally, we compare the results with the ones obtained with the multitask approach. These results are reported in Table 3.1 for all versions of the datasets. The results show how the multiclass-only classification model has the better performance when trained on all the available data, with $\sim$34% higher F1-score. In the same way, multitask classifier shows $\sim$10% higher performance, with a 79.19% F1 score in the unbalanced data scenario. From these results we may confirm that the amount of available data plays a key role in the vulnerability classification task and that a greater amount of data is required to achieve better results.

TABLE 3.1: Multiclass classification results

Setting	Accuracy	Precision	Recall	F1
Multiclass (only vulnerable)	53.31	55.68	56.43	55.39
Multiclass (balanced)	70.87	68.06	69.18	68.15
Multiclass (unbalanced)	79.61	77.92	81.81	79.67
Multitask (balanced)	71.86	69.85	72.37	70.79
Multitask (unbalanced)	79.03	77.39	81.40	79.19

TABLE 3.2: Comparison between LineVul and multitask binary classification

Setting	Accuracy	Precision	Recall	F1
LineVul (balanced)	97.81	99.39	96.01	97.67
Multitask (balanced)	97.88	97.9	97.94	97.88
LineVul (unbalanced)	99.07	96.89	87.2	91.79
Multitask (unbalanced)	99.03	97.31	93.87	95.51

For the binary task, we compare the multitask model with LineVul, retrained on both version of the dataset. The results are shown in table 3.2. It is noticeable, in the unbalanced scenario, that the proposed model preserves the LineVul performance, with the exception of a higher recall value. In this case, recall indicates the rate of function correctly classified as vulnerable with respect of all vulnerable functions. So, a higher recall value may indicate that the model has gained a higher coverage of the vulnerable class. The models trained on balanced data show the same overall performance.

RQ2. What are the effects of the multitask approach in vulnerability localization?

For this RQ we leverage the attention-based approach proposed by Fu et al., computing the attention scores assigned to every line of code and evaluating them to identify the ones that represent the causes of vulnerability. We evaluate the model performance in line-level localization using the model agnostic techniques introduced by Fu et al.: Layer Integrated Gradient (LIG) [43], Saliency [42], DeepLift [2], [41], DeepLiftSHAP [33], GradientSHAP [33]. This choice is justified by the logic behind the mechanism adopted for the line-level detection task, that leverages model explainability concepts to identify the most relevant features in the prediction of the functions vulnerability status (vulnerable or not vulnerable). The benefit from this approach is dual: it shows the high transformers performance in a line level task and the attention mechanism potential as a model explainability tool.

Replicating the tests, the model registered a 63% top10 Accuracy and an IFA value of 5.22 with median 2. Compared with LineVul our model has slightly worse performance. This is explained by the task the models are trained for. LineVul is trained exclusively for vulnerability detection, so it assigns the higher attention score to the tokens which better help it detecting the vulnerability. Our model is trained instead for vulnerability detection and vulnerability classification, so it needs to assign the attention scores considering the tokens that help it classify the vulnerability. Another aspect that affect the line-level results is the true positive rate. Considering that only the vulnerable functions have a flaw-line and flaw-line-index in the

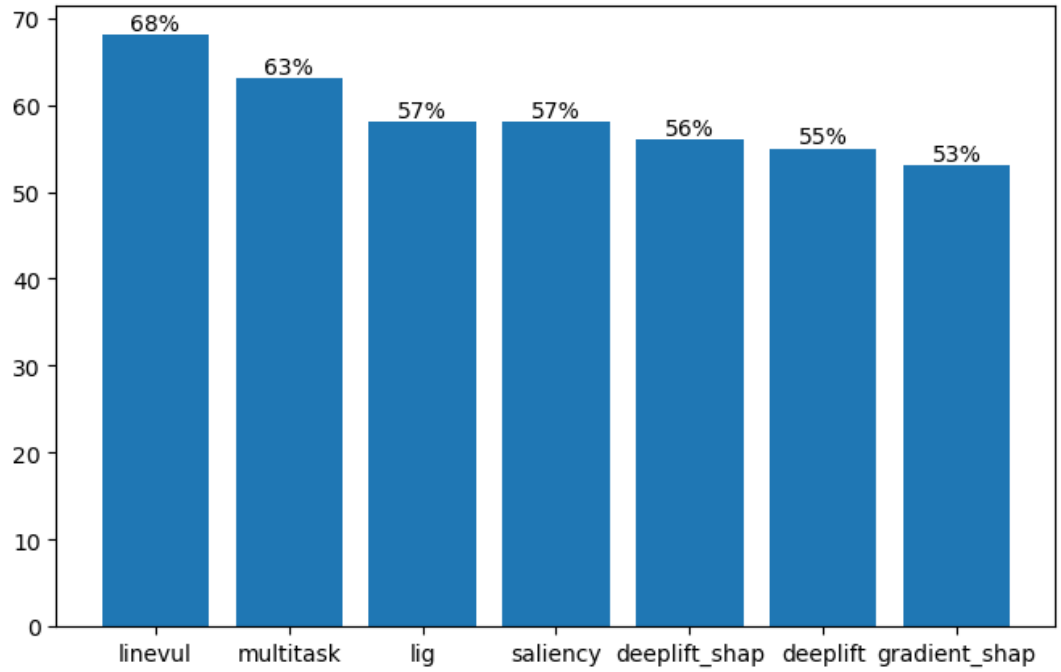


FIGURE 3.2: Top-10 accuracy computed with attention mechanism and model agnostic techniques compared

dataset, the line-level evaluation is performed only on the model's true positive outputs. Recall values in RQ1 indicate that our model achieved a higher true positive rate than LineVul, consisting in a higher number of samples evaluated. Therefore, this could be another possible cause of lower performance. However, the results are still better than the ones of the model agnostic explainability tools, as shown in Figures 3.2 and 3.3.

RQ3. Does a weighted loss improve the model's performance compared to using an average loss

All the previous evaluations have been done computing the average loss between multiclass and binary classification ones. With this RQ we want to study a different approach to compute the loss of the model. In this case, we run multiple experiments evaluating how a weighted loss may have an impact on model performance. Particularly, from the previous experiments it has been observed that the multiclass task is the one with the higher loss values when compared with the binary one, as shown in Figure 3.4.

The binary loss assumes values between 0.07 and 0.05, while the multiclass one has a value of 1.86 at epoch 0 and a value of 0.75 at epoch 9. From this observation, we decided to perform new tests with both the datasets leveraging different weight values for the multitask loss, keeping the value of 1.0 for the binary one. The obtained results, however, do not show significant performance improvements. Registered F1-scores presents slightly different values when compared with the ones obtained with the previous strategy. Further tests are necessary to explore new potentially ways to handle the high loss gap between our tasks.

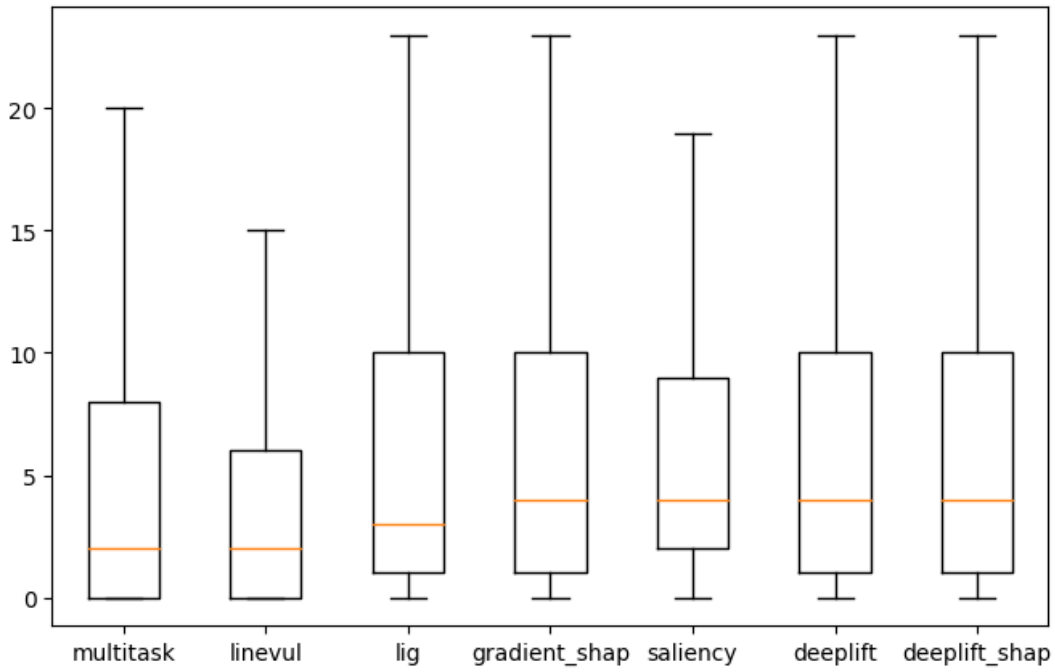


FIGURE 3.3: Initial False Alarm (IFA) computed with attention mechanism and model agnostic techniques compared

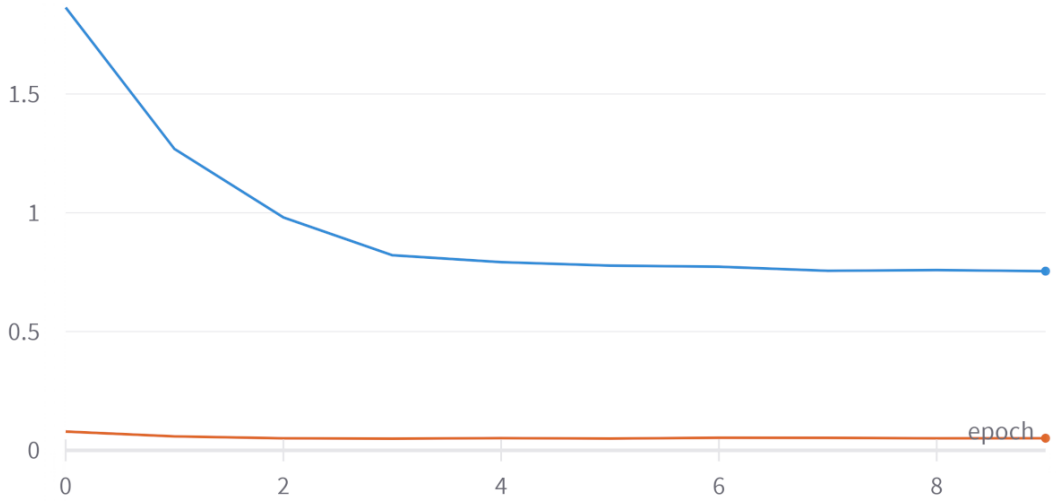


FIGURE 3.4: Binary loss (orange) and multiclass loss (blue) computed during evaluation with unbalanced data

3.6 Results and Discussion

From the performed studies we obtained the following results:

1. The multitask approach outperforms the multiclass-only one on the balanced dataset, while the ones with unbalanced data share almost the same performance. Both apporaches show the best results when trained with an higher number of samples, even if highly unbalanced;

2. The binary tasks show better performance when trained on the balanced dataset. LineVul and our model share similar results in vulnerability detection at function-level; in the unbalanced data scenario, the higher recall value indicates that the multitask approach enhanced the true positive rate relatively to the false negative rate, that results in a lower number of vulnerable function predicted as not-vulnerable.

3. Our model has higher line-level localization performance when compared with the other benchmark techniques, but slightly worse than LineVul.

4. A weighted approach to loss functions handling has near zero impact on the model's performance.

As pointed out by this study, a critical aspect of research in vulnerability classification is related in particular to the available data and some compromise needed to be done:

1. The amount of registered vulnerable functions is very limited, hence it is very difficult to get a big and balanced dataset.

2. Some vulnerabilities occur less frequently than others and there are few registered functions representative of these categories. For this reason it is difficult to learn a representative pattern for a lot of CWEs.

Another aspect to considered is the method adopted for line-level localization. The approach proposed by Fu et al. leverages a key aspect of the transformer-based architectures, achieving state-of-the-art performance. However, from our results, we see that this is tied with the task we are training the model. Training the model for vulnerability detection may make it assign the attention scores to the tokens that better help it to identify a vulnerability; but in our case the model needs to recognize the type of vulnerability too. Indeed, our results in line-level localization, even if better than the benchmark explainability methods, are lower than LineVul ones. As future work, we are interested in continuing the study of new ways to identify the vulnerable lines in the code for a more informative implementation of the vulnerability detector.

3.7 Conclusions

In this study we explored two possible transformer-based approaches for software vulnerability detection, extending the previous work by Fu et al. We performed two evaluations for both approaches to demonstrate the impact of the data used for the training on the models. We showed how using a more populated but unbalanced dataset produces the better results in vulnerability categorization, while a balanced

one produce better results in vulnerability detection. Furthermore, comparing our results with Fu et al., we assessed how the multitask approach is at the same time more informative for a security analyst and equally accurate on vulnerability detection at function-level. However, the subset of classes used is limited to vulnerability distribution in the BigVul dataset. We need to explore different grouping approaches for the vulnerability and more various dataset to get a wider coverage of the known flaws in source codes. A possible future work is to extend the capability of our model to cover a wider range of vulnerabilities or groups of vulnerabilities. Moreover, on line-level detection our model performed slightly worse than LineVul, but still better than the other benchmark techniques considered. Our goal is to explore new approaches to upgrade the model in this task. The findings presented in this chapter have been published in the proceedings of 21st International Conference on Security and Cryptography (SECRYPT 2024) [10].

Chapter 4

Can a Llama be a Watchdog? Exploring Llama 3 and Code Llama for Static Application Security Testing

4.1 Introduction

Many studies have focused on a wide variety of transformer-based architectures, ranging across all the different kinds of architectures (i.e., encoder-only, decoder-only and encoder-decoder) [30, 16]. Generally, research in vulnerability detection and categorization formalizes these problems as classification ones: binary classification for detection and multiclass classification for categorization. Different studies have delved in the vulnerability categorization problem, focusing on various approaches:

- pre-trained models fine-tuning [18]
- LLM zero-shot prompting [17]. The cited study, however, shows that the zero-shot approach is the one producing the least favorable outcomes, and a fine-tuning process is needed.

Despite classification being best suited for encoder-only transformers, it has been demonstrated that various architectures can achieve remarkable results in this kind of task [6]. Llama 3 [1], the latest version of Meta's foundation models, exemplifies this versatility. Pre-trained on 15 trillion tokens—seven times more than Llama 2 [45], with four times more code—Llama 3 shows significant improvements over its predecessor. The model demonstrates higher performance across different benchmarks, highlighting its potential for advanced tasks in static security analysis. Currently, however, there is no specialized version of Llama 3 for programming languages, in contrast with Code Llama [40], a family of large language models based on Llama 2. Code Llama provides state-of-the-art performance among open models, with infilling capabilities, support for large input contexts, and zero-shot instruction following ability for programming tasks. Despite the shown performance in different code manipulation tasks (e.g., code generation, infilling evaluation), Code Llama's performance in bug/vulnerability detection has not been fully explored. In this study, we explore Llama 3 and Code Llama capabilities in Static Application Security Testing related tasks, i.e., vulnerability detection, classification on two different benchmarks (BigVul [14] and DiverseVul [6]) and fine-grained localization on BigVul. Specifically, we explore the Meta AI models' suitability for static security

analysis tasks and whether the newest version, i.e., Llama 3, can surpass its programming language-specialized predecessor. Additionally, we test if the model's performance may be affected by a comment removing preprocessing. Given the nature of the attention-based flaw line detection approach, it is plausible that the presence of comments in the analyzed functions could mislead the model's predictions in identifying highly probable vulnerable lines. We compare our results with some state-of-the-art model widely used in vulnerability assessment CodeBERT [15], PolyCoder [49] and NatGen [5].

Our study's contribution can be summarized as follows:

- We provide a complete analysis of the newest Meta AI model, Llama 3, and its code-specialized predecessor, Code Llama, when trained on three SAST tasks;
- We explore the benefits derived from a simple data preprocessing procedure to enhance model performance in flaw line attention-based localization.

4.2 Background

4.2.1 Llama 3

Meta AI developed and released the Llama 3 family of large language models (LLMs) [1], a collection of pre-trained and instruction-tuned generative text models in 8B and 70B sizes. Llama 3 is an autoregressive language model that uses an optimized transformer architecture. The tuned versions use Supervised Fine-Tuning (SFT) and Reinforcement Learning with Human Feedback (RLHF) to align with human preferences for helpfulness and safety. Llama 3 was pre-trained on over 15 trillion tokens of data from publicly available sources. Specifically, Llama 3 was trained with 7 times the data used for training Llama 2 and 4 times the source code data. The fine-tuning data includes publicly available instruction datasets as well as over 10 million human-annotated examples.

4.2.2 Code Llama

Code Llama [40] is a family of large language models for source code by Meta AI, based on Llama 2. It includes three types of models, each type is released with 7B, 13B, 34B parameters. Their approach is based on gradually specializing and increasing the capabilities of Llama 2 models by applying a cascade of training and fine-tuning steps.

1. Foundations models (Code Llama) constitute the foundation models for code generation. The 7B and 13B models are trained using an infilling objective and are appropriate to be used in an IDE to complete code in the middle of a file, for example. The 34B model was trained without the infilling objective. All Code Llama models are initialized with Llama 2 model weights and trained on 500B tokens from a code-heavy dataset.

2. Python specialization (Code Llama-Python) are specialized for Python code generation. They are designed to study the performance of models tailored to a single programming language, compared to general-purpose code generation models. Code Llama Python models are further specialized on 100B tokens using a Python-heavy dataset. All Code Llama - Python models are trained without infilling and subsequently fine-tuned to handle long contexts.

3. Instruction-following models (Code Llama-instruct) are based on Code Llama and fine-tuned with additional 5B tokens to better follow human instructions.

4.3 Related Works

A study by [50] explored the effectiveness of various LLMs for vulnerability detection, including Code Llama. The research evaluated models using four paradigms: zero-shot learning, one-shot learning, discriminative fine-tuning, and generative fine-tuning. The study utilized two datasets: BV-LOC for C/C++ functions and SC-LOC for Solidity-based smart contracts. Among the tested models, CodeLlama-7B showed exceptional performance, particularly with discriminative fine-tuning, achieving significantly higher accuracy compared to traditional methods. This approach reframes Automated Vulnerability Localization (AVL) as a sequence labeling task, allowing precise identification of vulnerable code statements. Innovative strategies such as the sliding window and right-forward embedding were employed to overcome challenges related to input length and context, further enhancing the effectiveness of Code Llama. While zero-shot and one-shot learning offered some benefits, they were less effective than fine-tuning methods. Zhang et al.'s findings show the promise of Code Llama in AVL tasks, demonstrating high precision and improved performance through advanced fine-tuning techniques. However, the study also notes the need for further refinement to handle diverse vulnerability types and improve generalizability across different projects.

Building on these results, our work investigates Code Llama in a broader SAST context, comparing it with Llama 3 and assessing its performance not only on vulnerability localization but also on detection and classification tasks across multiple benchmarks. Unlike Zhang et al., who primarily focused on fine-tuning strategies and sequence labeling, we emphasize the evaluation of Code Llama's generalization capabilities and its comparison with other state-of-the-art LLMs under more diverse experimental settings. This complementary perspective provides a more comprehensive understanding of the strengths and limitations of Code Llama, highlighting both its potential and the open challenges in applying LLMs to real-world vulnerability detection scenarios.

4.4 Methodology

4.4.1 Models

In this study we compare two Llama models with three other state-of-the-art models, namely CodeBERT [15] – previously described in Chapter 3 –, PolyCoder [49] and NatGen [5].

PolyCoder

uses the same model architecture and pre-training objective as GPT-2, but pre-trains the model from scratch. The authors pre-trained the model with data from GitHub containing both source code and natural language comments within the code files. Their training data contains over 24K repositories in C/C++. The authors curate an

evaluation dataset of codes from unseen repositories during training. On the C programming language, PolyCoder achieves the lowest perplexity [1] value, compared to GPT-Neo, GPT-J, and Codex.

NatGen

Chakraborty et al.[5] proposes a new pre-training objective called "naturalizing" pre-training, similar to a code editing process. The authors generate un-natural code through transformations that preserve the code's semantics, including adding dead code, changing a while loop to a for loop without variable initialization, renaming variables, inserting confusing code elements, etc. Then, the pre-training objective asks the model to naturalize the code to the original developer-friendly form. Compared to CodeT5 [48], NatGen improves the performance over various downstream tasks such as code translation, text-to-code generation, and bug repair.

4.4.2 Dataset

We conducted an extensive comparison between Llama 3, Code Llama, CodeBERT, PolyCoder and NatGen on two C/C++ codebases, namely BigVul [14] – introduced in Chapter 3 – and DiverseVul [6].

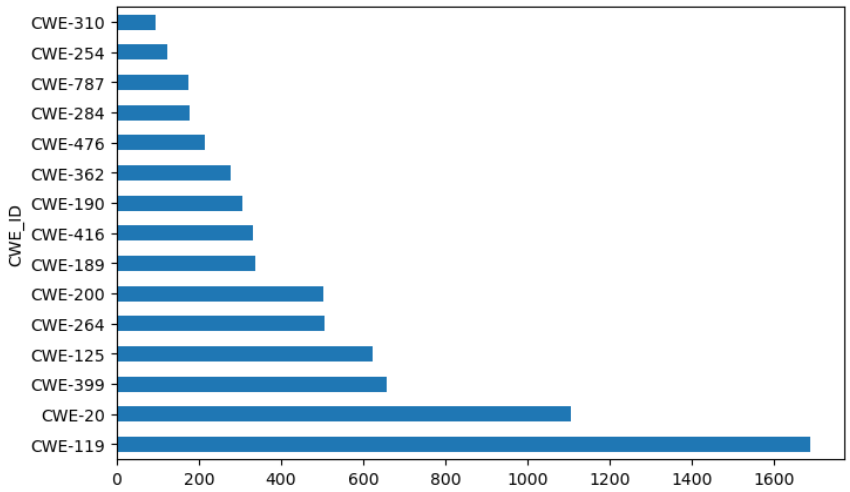
DiverseVul is a large C/C++ vulnerability dataset obtained by the following procedures: security issue websites crawling, vulnerability reports collection, vulnerability fixing commits extraction, and, for each vulnerability, cloning of the project and vulnerable and non-vulnerable source code extraction – a detailed description of the dataset and of the data gathering procedure can be found in Chapter 5 –. The result is a total of 18,945 vulnerable functions and 330,492 non-vulnerable functions, extracted from 7,514 commits and covering 150 CWEs. DiverseVul is one of the largest vulnerability codebases, built to provide more quality data for the training of large deep learning models. We obtained the dataset from the author's GitHub repository, [2]. As for BigVul, we performed a cleaning procedure on the data. As a matter of fact, all the dataset entries came with the CWE feature as a list of zero, one or more CWE IDs. For simplicity, we removed all the entries with more than one CWE ID, converted them as a string, and changed all the non-vulnerable entries to CWE feature with a "Not-vulnerable" string. As a result, we removed only 27 entries.

The main difference between the two datasets is that DiverseVul lacks flaw line information. Consequently, we are able to perform line-level evaluation only on BigVul data.

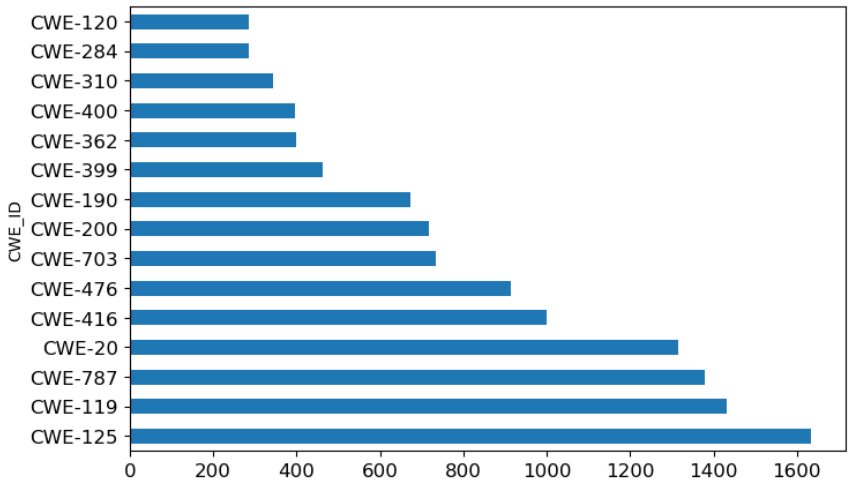
An important aspect of the datasets is their high imbalance among all the different CWE IDs, as shown in Figure 4.1a and 4.1b. This is related to the fact that some vulnerabilities may occur less frequently than others. To get more reliable results, we restrict the multiclass classification to the top 15 CWE classes in terms of distribution in the dataset. In the end, the final datasets are composed of 126,313 functions, with 7,089 vulnerable ones for BigVul and 198,090 functions, with 8,550 for DiverseVul. Considering the high imbalance between vulnerable and not vulnerable samples in both datasets, we test the model's performance by training it with a balanced version

[1] Perplexity is a standard metric for evaluating language models. It measures how well a model predicts a sequence of tokens: lower values indicate higher confidence and better performance. Formally, perplexity is defined as the exponential of the average negative log-likelihood of the tokens in a text sequence, thus reflecting the model's uncertainty.

[2] https://github.com/wagner-group/diversevul



(A) BigVul



(B) DiverseVul

FIGURE 4.1: Benchmark datasets top 15 CWE IDs distribution between vulnerable functions.

of them, obtained by undersampling the not vulnerable occurrences. The resulting balanced dataset is composed of 14,178 functions for BigVul, and 17,100 for DiverseVul, equally distributed between vulnerable and not vulnerable.

For all tests, we split the datasets into train, validation, and test sets with 80/10/10 ratio, with validation and test sets stratified to keep the same CWE ID class distribution as the train set.

4.4.3 Evaluation metrics

Accuracy, precision, recall and F1-score are computed for training, validation, and testing phases. For line-level detection, the models' performance is measured by Top-10 accuracy. Top-10 accuracy is computed by generating a top-10 ranking of lines for each function based on their cumulative attention scores, which are obtained by summing the attention scores of all tokens belonging to each line. We then

TABLE 4.1: Experimental results. The left section displays the classification F1-scores and line detection top-10 Accuracy measured on the BigVul dataset. The right section shows the classification results on the DiverseVul dataset. The best results for every task are highlighted in red.

	Bigvul				DiverseVul	
	Binary					
	Base source code		**Comment-less source code**		**Base source code**	**Comment-less source code**
	F1-score	**Top-10 Accuracy**	**F1-score**	**Top-10 Accuracy**	**F1-score**	**F1-score**
Llama 3	95,3	65,23	95,3	65,23	72,87	74,02
Code Llama	95,05	62,31	94,57	69,63	74,3	74,65
CodeBERT	94,16	71,04	94,07	75,4	73,21	73,09
PolyCoder	94,08	67,45	94,14	71,81	73,91	73,66
NatGen Encoder	94,26	67,35	94,32	72,57	75,2	74,39
	Multiclass					
	Base source code		**Comment-less source code**		**Base source code**	**Comment-less source code**
	F1-score	**Top-10 Accuracy**	**F1-score**	**Top-10 Accuracy**	**F1-score**	**F1-score**
Llama 3	59,85	63,52	61,17	63,52	55,69	57,19
Code Llama	62,62	62,34	63,98	66,1	60,36	59,49
CodeBERT	63,32	68,15	61,27	69	58,99	58,63
PolyCoder	47,05	63,21	44,18	66	45,35	44,24
NatGen Encoder	58,3	67,61	59,32	67,45	57,04	57,36
	Multiclass+NotVul					
	Base source code		**Comment-less source code**		**Base source code**	**Comment-less source code**
	F1-score	**Top-10 Accuracy**	**F1-score**	**Top-10 Accuracy**	**F1-score**	**F1-score**
Llama 3	63,33	63,52	59,48	63,52	53,74	51,35
Code Llama	64,69	60,19	65,08	67,51	55,91	55,29
CodeBERT	59,01	69,4	59,39	73,64	54,05	58,63
PolyCoder	47,3	65,9	49,16	71,75	43,03	43,32
NatGen Encoder	60,45	67,5	58,63	72,83	50,94	53,67

calculate the percentage of these functions in which at least one actual vulnerable line appears in the ranking.

4.5 Experiments Setting

We structure our evaluation into two tasks:

1. vulnerability detection, formalized as a binary classification,
2. vulnerability classification, formalized as a multiclass classification.

The classification head architecture is structured with a single linear layer with 2 output neurons in the binary scenario and 15 output neurons for the multiclass. We consider two possible approaches for the vulnerability classification task: classify only vulnerable functions and classify vulnerable and not vulnerable functions together. This is done considering “Not-vulnerable" as an additional class to classify with CWE IDs. In this way, we get two different multiclass classification tasks: classification of 15 different CWE classes and classification of 14 CWE classes and one “Not-vulnerable" class.

For the hyperparameters, we choose $2 * 10^{-5}$ for the learning rate, with AdamW as the optimizer and 512 sequence length for every model. To accommodate memory limitations, we adopt varying batch sizes for different model types. Specifically, we utilize a batch size of 32 for CodeBERT, PolyCoder, and Natgen, while opting for a batch size of 4 for Llama 3 and Code Llama. The models are trained for 10 epochs, with cross-entropy as the loss function for all the classifications. The training is performed on an NVIDIA RTX A6000 GPU.

4.5.1 Models Settings

For all the models, in the first step, we obtain the pre-trained weights and tokenizers from the HuggingFace repository [3].

Llama3 and Code Llama

Llama 3 and Code Llama are characterized by a multi-billion number of parameters (8B and 7B for the smaller versions, respectively). To be able to fine-tune these models with our hardware constraints, we need to reduce the number of trainable parameters. For this purpose, the HuggingFace library PEFT (Parameters Efficient Fine-Tuning) [35] offers a set of fine-tuning techniques to better optimize memory usage while training large models. LoRA (Low-Rank Adaptation) [22] is the most used one. In this study, we use a variant of LoRA, namely QLoRA (Quantized Low-Rank Adaptation) [12]. QLoRA takes LoRA a step further, quantizing the weights of the LoRA adapters to lower precision (e.g., 4-bit instead of 8-bit). This further reduces the memory footprint and storage requirements. Finally, a classification head is built.

CodeBERT and PolyCoder

CodeBERT and PolyCoder are respectively encoder-only with a 125M parameters transformer and a decoder-only 160M parameters transformer. Considering the fewer memory usages from both models, no PEFT techniques are used, and traditional fine-tuning is performed.

NatGen

NatGen is an encoder-decoder transformer with 220M parameters, based on T5 architecture [39]. The idea is to take advantage of the peculiar pre-training task adopted by NatGen when fine-tuned to a vulnerability detection task. Considering the nature of the tasks, we decide to leverage only NatGen encoder stacks, defining a custom model for sequence classification whose architecture is composed of a NatGen encoder module and a classification head.

4.6 Results and Findings

Our study evaluates the performance of five models—Llama 3, Code Llama, CodeBERT, PolyCoder, and NatGen—on the BigVul and DiverseVul datasets for binary and multiclass vulnerability classification tasks. Results are summarized in table 4.1.

The binary classification results show that both Llama 3 and Code Llama achieve high F1-scores on the BigVul dataset, with Llama 3 scoring slightly higher in F1-score (95.3%). On the DiverseVul dataset, Code Llama performs the best with an F1-score of 74.65%, indicating its robustness across different datasets. The variations in performance can be attributed to differences in the dataset characteristics and the model architectures. For instance, Code Llama, being pre-trained specifically on source code, shows a slight edge in handling source code data.

[3] https://huggingface.co/models

The multiclass classification results show that Code Llama outperforms the other models in F1-score in the comment-less scenario on the BigVul dataset, while it achieves the best results on DiverseVul. Furthermore, Code Llama achieves the best results in multiclass classification with the “Not-Vulnerable" class, except in the comment-less scenario.

CodeBERT also performs well, especially in the base source code scenario and in all the Top-10 Accuracy results.

Removing comments from the source code before training generally leads to improved performance. For example, Code Llama’s F1-score increased from 62.62% to 63.98% on the BigVul dataset after comments were removed. This suggests that comments might introduce noise, affecting the model’s ability to focus on the relevant code features for vulnerability detection.

The results demonstrate that Code Llama and Llama 3 are highly effective in vulnerability detection and classification tasks, with Code Llama showing a slight advantage due to its specialized training on source code. The high F1-scores and Top-10 Accuracy indicate that these models can reliably identify vulnerabilities, making them suitable for SAST.

Despite the promising results, this study has limitations. The primary limitation is the reliance on balanced datasets, which may not fully represent real-world scenarios where vulnerable functions are rarer. Future work should explore the models’ performance on unbalanced datasets and investigate more advanced techniques for handling class imbalance. Additionally, expanding the evaluation to include other programming languages and more diverse datasets would provide a more comprehensive understanding of the models’ capabilities.

4.7 Conclusions

In this study, we explored Llama 3 and Code Llama capabilities in SAST and compared their results with state-of-the-art code analysis models. Our results demonstrate that Code Llama and Llama 3 are highly effective in vulnerability detection and classification tasks, with Code Llama showing a slight advantage due to its specialized training on source code. The high F1-scores and Top-10 Accuracy indicate that these models can reliably identify vulnerabilities, making them suitable for SAST. Moreover, the study shows how our comment cleaning approach enhances the line localization performance without degrading the downstream tasks results for all the evaluated models. These results have been discusses during the 2024 IEEE International Conference on Cyber Security and Resilience (CSR) conference and published in its proceedings [9].

Chapter 5

Addressing the C/C++ Vulnerability Datasets Limitation: the Good, the Bad and the Ugly

5.1 Introduction

Recent advancements in software vulnerability prediction (SVP) have demonstrated that high-quality datasets are essential for training effective machine learning models. Studies have highlighted significant challenges in dataset quality, including inaccurate labels, data duplication, and limited diversity in vulnerability types [7]. For instance, Croft et al. revealed that up to 71% of labels in real-world datasets are inaccurate, which undermines the performance and generalization of models trained on such data. Furthermore, while synthetic datasets like Juliet[1] provide an additional source of training data, they often lack the complexity and variability of real-world vulnerabilities. This has led to an increased focus on real-world datasets [14], [6], [38], which, despite their inherent imperfections, offer a more realistic foundation for training models capable of detecting vulnerabilities in diverse codebases.

One of the main limitations in applying large language models (LLMs) to vulnerability detection is the significant data requirements for training. Beyond the volume of data, the origin of the data, whether synthetic or real-world, plays a critical role in shaping model performance [4]. Real-world datasets, derived from actual software projects, are particularly valuable for evaluating the generalization capabilities of models. However, achieving robust generalization remains a significant challenge. Generalization in this context refers to a model's ability to detect vulnerabilities across unseen projects, novel vulnerability types (e.g., new Common Weakness Enumeration, CWEs), and varied data distributions. High performance on specific datasets often masks the inability of models to generalize effectively, an issue that has been underexplored in the literature.

This study bridges existing gaps by conducting an in-depth evaluation of three widely-used C/C++ vulnerable code datasets. Using a RoBERTa-based model, we examine its generalization capabilities across three key scenarios: unseen projects, novel vulnerability types, and varying data distributions. Furthermore, we analyze the effects of aggregating these datasets into a unified collection to assess its impact on model performance.

[1] https://samate.nist.gov/SARD/test-suites/112

Our results reveal that dataset aggregation significantly improves model generalization, particularly in challenging cases involving unseen projects and previously unencountered vulnerabilities. This finding emphasizes the value of combining datasets to mitigate biases and enhance the robustness of deep learning models in software vulnerability detection.

By addressing dataset limitations and exploring their role in transformer-based models, this work underscores the importance of diverse, high-quality datasets as a foundation for advancing automated software security assessment. The insights presented aim to guide researchers and practitioners in developing scalable and reliable solutions for software vulnerability detection.

5.2 Vulnerability Datasets Quality

The quality of vulnerability datasets is a crucial, often overlooked, factor in the effectiveness of machine learning models for software security [7]. While transformer-based models show high performance on specific datasets, their ability to generalize to diverse, unseen data remains a challenge [6]. Most studies focus on performance metrics within specific datasets [53], neglecting how models handle real-world data distributions.

This emphasis on dataset-specific performance can give a false sense of model effectiveness. High accuracy on a particular dataset may mask weaknesses when confronted with new vulnerabilities or projects. This issue is exacerbated when datasets have biases in the types of vulnerabilities, code structure, or programming languages represented. In practice, these biases may arise from limited sources of data collection, over-representation of certain CWE categories, or reliance on benchmark datasets that do not reflect the heterogeneity of real-world software projects.

Another key issue is the presence of noise and label inconsistencies. Vulnerability datasets are often constructed through automated mining of repositories, vulnerability databases, or commit histories. While efficient, these processes can introduce inaccuracies such as mislabeled samples, incomplete context for vulnerable functions, or ambiguous mappings between vulnerabilities and code fragments. Such noise can mislead learning algorithms, inflating reported results while degrading real-world applicability.

Furthermore, dataset size alone does not guarantee quality. Large datasets may still lack representativeness if they predominantly capture a narrow subset of vulnerabilities, for example focusing mainly on memory-related issues in C/C++ while neglecting logic errors, concurrency bugs, or vulnerabilities in other programming languages. Similarly, many datasets fail to include non-vulnerable but security-relevant code, making it harder for models to learn fine-grained distinctions between safe and unsafe implementations.

To address these challenges, it is essential to focus on both dataset diversity and generalization. Vulnerability datasets should not only be large and comprehensive but also varied enough to represent real-world applications. This diversity is crucial for training models capable of detecting vulnerabilities across different codebases, vulnerability types, and project configurations. Promising directions include combining multiple datasets to reduce biases, curating balanced distributions of vulnerability categories, and incorporating code from a variety of domains and development practices. In addition, community-driven efforts toward standardized evaluation benchmarks could help mitigate the current fragmentation and provide a clearer picture of model robustness in realistic settings.

Ultimately, improving dataset quality is not a secondary task but a foundational requirement for advancing machine learning in software vulnerability detection. Without reliable, diverse, and representative datasets, even the most sophisticated models risk overfitting to artificial patterns, limiting their contribution to practical cybersecurity.

5.3 Related Works

The importance of dataset diversity and high-quality data was emphasized by other works [52], which suggested that aggregating multiple datasets could alleviate data bias and improve model robustness and generalization. Recently, [3] addressed critical limitations in existing datasets used for evaluating deep learning-based vulnerability detection models. They introduce the RealVul dataset, designed to represent realistic usage scenarios by including complete codebases rather than focusing solely on isolated snippets from fixing commits, as seen in prior datasets like BigVul and SARD [37]. They highlight significant discrepancies between the performance of deep learing models on synthetic or limited datasets and their practical application in real-world scenarios. Furthermore, they identify overfitting as a primary issue and propose an augmentation technique to improve generalization. This study underscores the necessity of realistic datasets for evaluating and improving the robustness of vulnerability detection models. However, despite its potential, RealVul was not included in our study. A major limitation lies in the size of its functions, which are significantly larger than those contained in the datasets we evaluated. This characteristic would have resulted in considerably longer training times and resource demands, making direct comparison impractical within the scope of our experiments. Consequently, we focused our analysis on datasets that allowed for more feasible training and evaluation, while recognizing that incorporating larger and more realistic datasets like RealVul represents an important direction for future work.

Additionally, synthetic datasets, like Juliet, have been used to augment training data, though these datasets may lack the complexity of real-world vulnerabilities. Our study extends this body of work by evaluating the impact of real-world vulnerability datasets on transformer-based models, examining the importance of diverse data distributions for improving vulnerability detection and model's generalization through various experimental setups.

5.4 Methodology

5.4.1 Datasets

This section offers a comprehensive overview of the analyzed datasets, detailing their key characteristics and the methodologies employed by the authors in their construction. To facilitate a clearer understanding of the datasets' features, a concise summary of each data collection and processing procedure is presented.

BigVul

BigVul [14] is a large C/C++ vulnerability dataset, collected from open-source GitHub projects. Big-Vul dataset contains the details of CVE entries from 2002 to 2019 and covers 358 different projects that are linked to 4,432 unique code commits. The 4,432

code commits contain the code fixes for 3,754 vulnerabilities in 91 Common Weakness Enumeration (CWE) types. We obtained the current version of BigVul from the official GitHub repository [2].

The Big-Vul dataset was constructed through a systematic five-step process.

1. **CVE data crawling**: vulnerability entries were crawled from the CVE database using a BeautifulSoup2 based script, collecting detailed information such as CVE ID, summary, CWE ID, and security impact metrics.

2. **GitHub reference links selection**: CVE entries with reference links to publicly available Git repositories were selected, focusing on repositories with clear paths to code commits, such as Github and proprietary Git servers for popular products like Google Android and Chrome.

3. **Code commits extraction**: unique crawling strategies were implemented to extract bug IDs and map them to relevant code commits, ensuring linkage between CVEs and their associated code changes.

4. **Commit histories analysis**: commit histories were analyzed to identify vulnerable and fixed versions of methods within projects.

5. **Dataset construction**: the data set was structured into CSV files that contain comprehensive project information and zipped packages of source code functions.

This approach ensures a precise mapping between vulnerabilities, code fixes, and project metadata.

DiverseVul

DiverseVul [6] is another large C/C++ vulnerability dataset obtained by the following procedures: crawling of security issue websites, collection of vulnerability reports, extraction of vulnerability fixing commits, and, for each vulnerability, cloning of the project and extraction of vulnerable and non-vulnerable source code. The result is a total of 16,109 vulnerable functions and 311,560 non-vulnerable functions, extracted from 7,514 commits and covering 150 CWEs. DiverseVul is one of the largest vulnerability codebases, built to provide more quality data for the training of large deep learning models. We obtained the dataset from the author's GitHub repository, [3]. As for BigVul, we performed a cleaning procedure on the data. As a matter of fact, all the dataset entries came with the CWE feature as a list of zero, one, or more CWE IDs. For simplicity, we removed all the entries with more than one CWE ID, converted them as a string and all the entries without a CWE ID, represented as '[]' in the table. As a result, we removed 45,922 entries. The resulting dataset is composed of 16,109 vulnerable functions and 265,638 not vulnerable functions.

The collection of high-quality vulnerability-fixing commits for DiverseVul involved multiple steps.

1. **Selction of Security issue websites**: The process began with identifying 29 security issue websites, narrowing it down to two with the most Git system commits. From these, issue titles, bodies, and Git commit URLs were crawled.

[2] https://github.com/ZeoVan/MSR_20_Code_vulnerability_CSV_Dataset
[3] https://github.com/wagner-group/diversevul

2. **Exclusion of vulnerability-Introducing commits**: To exclude vulnerability-introducing commits, two heuristics were applied: filtering out commits referenced in discussions with keywords like "introduced" and manually excluding commits modifying 10 or more functions if they introduced vulnerabilities.

3. **Commit Parsing and Dataset Construction**: Details such as project information and commit IDs were extracted from the filtered commits. Projects were cloned, and their commits were analyzed to identify relevant C/C++ code files. Functions in these files were categorized into three groups: Vulnerable

   (a) functions (pre-fix state).

   (b) Non-vulnerable functions (post-fix state).

   (c) Unchanged non-vulnerable functions.

   Deduplication was performed using MD5 hashes, prioritizing the inclusion of vulnerable functions. The resulting dataset comprised 7,514 commits from 797 projects, containing 18,945 vulnerable functions and 330,492 non-vulnerable functions spanning 150 CWEs. The remaining commits were parsed to extract project details and commit IDs. The projects were cloned, and commits were analyzed to identify C/C++ code files. Functions in these files were labeled as vulnerable (pre-fix), non-vulnerable (post-fix), or non-vulnerable unchanged functions. Deduplication was performed using MD5 hashes, prioritizing vulnerable functions. Ultimately, the dataset included 7,514 commits from 797 projects, yielding 18,945 vulnerable functions and 330,492 non-vulnerable functions across 150 CWEs.

4. **Commits referencing CVE numbers**: the National Vulnerability Database (NVD) API was used to retrieve CWE mappings. Developer-annotated vulnerabilities were manually categorized into 25 widely recognized CWE classes. Approximately 85% of the dataset was successfully mapped to CWEs, with some CVEs associated with multiple categories.The dataset demonstrated a broad diversity of projects and vulnerabilities. Notably, CWE-703 ("Improper Check or Handling of Exceptional Conditions") appeared frequently, despite not being included in MITRE's top-25 CWE list. For issues mentioning CVE numbers, the National Vulnerability Database API was queried to obtain CWE information. Developer-annotated vulnerabilities were manually mapped to 25 popular CWE categories. About 85% of the dataset was successfully mapped to CWEs, with some CVEs linked to multiple CWEs. The dataset highlights the diversity of projects and vulnerabilities, with CWE-703 ("Improper Check or Handling of Exceptional Conditions") appearing prominently despite not being in MITRE's top-25 CWEs.

MegaVul

MegaVul [38] is a comprehensive and continually updated dataset developed to support vulnerability detection in C/C++ software. MegaVul aggregates data from the Common Vulnerabilities and Exposures (CVE) database and associated Git-based repositories to address the limitations of previous datasets, including incomplete data, outdated entries, and limited diversity. The dataset encompasses 17,380 vulnerabilities extracted from 992 repositories and spans 169 vulnerability types, covering the period from January 2006 to October 2023. Unlike prior datasets, MegaVul

is designed for continuous updates, ensuring its relevance in vulnerability research. We obtained the dataset from the author's GitHub repository, [4].

The MegaVul dataset was developed through a comprehensive multi-step process to ensure high-quality, enriched data for vulnerability detection and modeling.

1. **Git Commit URL Mining**: Data collection began by crawling the National Vulnerability Database (NVD) for CVE entries, extracting descriptive information and reference links. A manual analysis identified 28 Git-based platforms frequently linked to CVEs. Methods such as parsing issue events, comments, and pull requests were employed to mine commit URLs.
2. **Git Commit URL Mining**: Data collection began by crawling the National Vulnerability Database (NVD) for CVE entries, extracting descriptive information and reference links. A manual analysis identified 28 Git-based platforms frequently linked to CVEs. Methods such as parsing issue events, comments, and pull requests were employed to mine commit URLs.
3. **Commit Download**: Vulnerability-fixing commits were downloaded, with C/C++ source codes and headers being prioritized. Platforms like GitHub and GitWeb were categorized, and platform-specific scripts were implemented for efficient data extraction, ensuring minimal irrelevant data was included.
4. **Function Extraction**: Functions within source files were separated using the syntax-aware parser Tree-sitter[5], enhanced to handle macros and complex syntax. This approach ensured accurate extraction, avoiding errors common in simpler methods like regular expressions.
5. **Data Distillation**: Various filters were applied to clean the dataset. Deduplication filters removed redundant functions and commit copies across datasets. Anomaly filters excluded templated codes and large-scale changes linked to version bumps or refactoring. Test-related and label-consistency filters ensured reliable annotations and prevented contradictions.
6. **Data Enrichment**: MegaVul was enriched with multiple dimensions of function representation, including function signatures, abstraction to address vocabulary explosion, static parsing for structures like ASTs and PDGs, and detailed code changes captured via diff tools. These representations were designed to support advanced modeling techniques.

The resulting dataset offers a robust foundation for vulnerability detection, combining rigorously filtered, accurately labeled data with enriched representations to cater to diverse analytical and modeling needs.

5.4.2 Similarities and Differences Between the Dataset Construction Procedures

Similarities

- **Use of CVE Data as a Source**: All three datasets rely on CVE data to identify vulnerabilities, ensuring standardized and reliable starting points for collecting information about vulnerabilities.

[4] https://github.com/Icyrockton/MegaVul

[5] Tree-sitter is a parser generator tool and an incremental parsing library. It can build a concrete syntax tree for a source file and efficiently update the syntax tree as the source file is edited. https://tree-sitter.github.io/tree-sitter/

- Focus on Git-Based Repositories: Each dataset includes Git repositories as a core source of code changes. Methods to mine or crawl Git repositories (e.g., GitHub, proprietary Git servers) are central to their construction.
- **Deduplication and Data Cleaning**: All datasets incorporate deduplication to remove redundant entries and apply filters to enhance data quality. These steps ensure the reliability and usability of the datasets for modeling purposes.
- **Inclusion of Function-Level Data**: Each dataset extracts functions from source code, emphasizing the granular level of vulnerable and non-vulnerable functions for precise vulnerability analysis.
- **Categorization and Enrichment**: All datasets include some form of categorization or enrichment, whether by mapping vulnerabilities to CWE categories or augmenting function data with structural representations like ASTs and PDGs.

Differences

- **Data Source Scope**:
  - Big-Vul relies on CVE entries with reference links to public repositories, focusing on a clear linkage between CVEs and code commits.
  - DiverseVul goes beyond CVE data, also crawling general security issue websites to collect vulnerability-fixing commits, ensuring broader project diversity.
  - MegaVul prioritizes Git-based platforms linked to CVEs and employs extensive manual analysis and tailored scripts to cover 28 platforms comprehensively.
- **Heuristics for Filtering Data**
  - DiverseVul applies specific heuristics to exclude commits introducing vulnerabilities, such as filtering based on keywords and manual reviews.
  - MegaVul introduces anomaly and consistency filters to refine its dataset further, addressing issues like version bumps and test-related contradictions.
  - Big-Vul does not describe such explicit heuristics, but relies on mapping bug IDs and analyzing commit histories.
- **Function Extraction Techniques**
  - MegaVul uses advanced tools like Tree-sitter to ensure syntax-aware parsing, addressing challenges with macros and complex syntax.
  - DiverseVul and Big-Vul do not specify such sophisticated parsing techniques but focus on extracting and categorizing functions.
- **Enrichment and Representation**
  - MegaVul emphasizes extensive enrichment, including representations like ASTs, PDGs, and detailed diffs for supporting advanced modeling techniques.
  - DiverseVul enriches its dataset with CWE mappings, focusing on vulnerability diversity and coverage.

  - Big-Vul structures its data into CSV files and provides zipped source code, prioritizing ease of access and organization.

- **Diversity of Sources and Projects**
  - DiverseVul highlights its effort to include a broader diversity of projects and vulnerability types by leveraging multiple websites and excluding vulnerability-introducing commits.
  - Big-Vul and MegaVul are more focused on CVE-linked repositories, with less emphasis on capturing broader project diversity.

- **Final Dataset Composition**
  - DiverseVul explicitly categorizes functions into three groups (vulnerable, post-fix non-vulnerable, and unchanged non-vulnerable).
  - MegaVul enriches its dataset with multi-dimensional representations to support advanced machine learning.
  - Big-Vul structures its dataset into CSV files and source code archives, focusing on accessibility.

Summary

While all three datasets share a reliance on CVE data, Git repositories, and function-level extraction, they differ in scope, methodologies, and the level of data enrichment. Big-Vul emphasizes clear mappings and structured organization, DiverseVul targets diversity and careful filtering, and MegaVul provides the most comprehensive enrichment for advanced modeling needs.

5.4.3 Model

We used a RoBERTa-like model designed for working with source code, namely CodeBERTa[6], trained on the CodeSearchNet dataset [23] from GitHub. Since the tokenizer is optimized for code rather than natural language, it represents the data more efficiently, reducing the length of the sequences by 33% to 50% compared to standard tokenizers such as those for GPT-2 or RoBERTa. The model itself is relatively compact, with six layers and 84 million parameters, similar in size to DistilBERT. It was trained from scratch on the entire dataset for five epochs.

5.4.4 Evaluation metrics

Considering the high imbalance in all three datasets,the Area Under the Receiver Operating Characteristic Curve (AUC) is preferred over accuracy, providing a more comprehensive and nuanced evaluation of a model's performance. Alongside AUC, precision, recall, and F1-score are used to assess specific aspects of performance. Precision focuses on minimizing false positives, which is critical for reducing unnecessary remediation efforts, while recall ensures most vulnerabilities are detected, avoiding missed security risks. The F1-score balances these two, offering a single metric that accounts for both. Additionally, False Positive Rate (FPR) and False Negative Rate (FNR) are included to directly measure errors; a low FPR minimizes false

[6] https://huggingface.co/huggingface/CodeBERTa-small-v1

alarms, and a low FNR ensures critical vulnerabilities are not overlooked. This combination of metrics ensures a holistic evaluation, capturing both accuracy and the trade-offs necessary for real-world applications.

5.5 Experiments Setting

The experiments were conducted to evaluate vulnerability detection in source code as a binary classification task across various generalization scenarios:

- **Random Split Benchmarking**: Training, validation, and testing were performed on randomly split subsets of the dataset to establish baseline performance.
- **Project Generalization**: Models were trained and validated on 90% of the source code from specific projects and tested on entirely different projects not included in the training set. This setup examines the model's ability to generalize across unseen projects, as prior studies indicate performance drops in such scenarios.
- **Vulnerability Type Generalization**: Training and validation were conducted on functions associated with vulnerabilities outside the MITRE Top 25 most dangerous CWEs. Testing was performed on data within the Top 25, focusing on how dataset vulnerability distributions impact model performance when detecting high-priority unseen vulnerabilities.
- **Cross-Dataset Evaluation**: Training and validation were performed on one dataset, followed by testing on a combined dataset from the other datasets included in the study.

These settings assess model performance under diverse and realistic conditions, emphasizing generalization across projects, vulnerability types, and datasets. For the hyperparameters, we choose $2 * 10^{-5}$ for the learning rate, with AdamW as the optimizer and 512 sequence length for every model. The training is performed on an NVIDIA RTX A6000 GPU.

5.6 Results and Discussion

The experimental results obtained from the three state-of-the-art datasets are presented in Table 5.1. Table 5.2 summarizes the performance achieved on the merged dataset across the three analyzed scenarios. Finally, Table 5.3 illustrates the model's generalization capability when trained on the merged dataset and evaluated on previously unseen data from the three original datasets.

5.6.1 The Good

CodeBERTa has shown excellent performance under favorable conditions, particularly when trained and tested on random splits of the same dataset. This scenario, often used as a baseline, allows the model to benefit from shared patterns and characteristics within the data. For example, on the BigVul dataset, the model achieved an impressive AUC of 92.24, coupled with a Precision of 81.22 and an F1-score of 83.39. These results demonstrate the model's capacity to effectively detect vulnerabilities when the distribution of the training and test data is consistent. Even on

TABLE 5.1: CodeBERTa performance on the different tested use cases. Values in parentheses indicate the metrics' differences with respect to the "Random Splits" case. Drop/Change values are highlighted in bold on a separate line.

Test Case	AUC	Precision	Recall	F1-score	FPR	FNR
Random Splits						
BigVul	92,24	81,22	85,69	83,39	1,22	14,31
DiverseVul	66,28	21,65	41,71	28,5	9,16	58,29
Megavul	71,38	56,4	44,61	49,81	1,85	55,39
Unseen Projects						
BigVul	86	88,7	72,65	79,88	0,64	27,35
(Drop / Change)	**-6,24**	**+7,48**	**-13,04**	**-3,51**	**-0,58**	**+13,04**
DiverseVul	53,39	15,71	10,79	12,79	4,01	8,35
(Drop / Change)	**-12,67**	**-9,02**	**-27,32**	**-17,21**	**-1,99**	**-53,94**
Megavul	58,03	39,46	18,54	25,22	2,47	81,46
(Drop / Change)	**-13,35**	**-16,94**	**-26,07**	**-24,59**	**+0,62**	**+26,07**
Unseen CWEs						
BigVul	88,45	84,4	77,86	81	0,97	22,14
(Drop / Change)	**-3,79**	**+3,18**	**-7,83**	**-2,39**	**-0,25**	**+7,83**
DiverseVul	56,59	17,44	18,67	18,03	5,5	81,33
(Drop / Change)	**-9,69**	**-4,21**	**-24,04**	**-10,47**	**-3,66**	**+23,04**
Megavul	59,4	29,68	21,77	25,12	2,96	78,23
(Drop / Change)	**-11,98**	**-26,72**	**-22,84**	**-24,69**	**+1,11**	**+22,84**
Unseen Datasets						
BigVul	53,94	39,47	8,63	14,16	0,75	91,37
(Drop / Change)	**-38,3**	**-41,75**	**-77,06**	**-69,23**	**-0,47**	**+77,06**
DiverseVul	69,23	53,85	40,42	46,18	1,95	59,58
(Drop / Change)	**+2,95**	**+32,2**	**-1,29**	**+17,68**	**-7,21**	**-7,21**
Megavul	72,58	30,75	52,35	38,75	7,18	47,65
(Drop / Change)	**+1,2**	**-25,65**	**+7,74**	**-11,06**	**+5,33**	**-7,74**

a more challenging dataset like Megavul, where diversity and noise are higher, the model maintained solid performance, with an AUC of 71.38 and an F1-score of 49.81, showing its robustness under less ideal conditions.

Beyond single-dataset experiments, combining the three datasets (BigVul, DiverseVul, and Megavul) into a unified training set further enhanced the model's performance, particularly on datasets that initially posed challenges. The DiverseVul dataset is a notable example: by merging data and removing duplicates, the model's F1-score increased by 6.18 points, rising from 28.5 to 34.68. This improvement highlights the potential benefits of addressing data sparsity and increasing the diversity of training samples. Combining datasets, therefore, emerges as a promising strategy for handling underrepresented vulnerabilities and improving overall model robustness.

5.6.2 The Bad

Despite these strengths, the model exhibited significant weaknesses when tested on unseen projects. These tests aimed to evaluate CodeBERTa's ability to generalize

TABLE 5.2: CodeBERTa performance when trained with all the data on three test cases. Values in parentheses indicate the drop or increase relative to the standard splits.

Test Case	AUC	Precision	Recall	F1-score	FPR	FNR
Standard splits	80,16	46,12	64,8	53,89	4,49	35,2
Unseen Projects	58,9	35,1	20,84	26,16	3,03	79,16
(Drop / Change)	**-21,26**	**-11,02**	**-43,96**	**-27,73**	**-1,46**	**+43,96**
Unseen CWEs	66,18	32,18	37,15	34,48	4,8	62,85
(Drop / Change)	**-13,98**	**-13,94**	**-27,65**	**-19,41**	**+0,31**	**+27,65**

beyond the specific patterns and styles of the training data. The results, however, revealed limitations. On the BigVul dataset, the F1-score decreased by 3.51 points, dropping from 83.39 to 79.88. While the decline may seem moderate, it reflects a diminished ability to handle new coding styles or project-specific conventions. For DiverseVul, the drop was much more severe: the F1-score fell by 17.21 points, plunging from 28.5 to 12.79. This suggests that the model struggles to adapt to the unique characteristics of different projects, which can vary widely in structure and context.

Testing on vulnerabilities with unseen CWEs revealed a similar trend. Vulnerabilities in this setting often involve patterns or characteristics that the model has not encountered before. Consequently, the performance declined sharply across metrics. In the Megavul dataset, for instance, the AUC dropped to 59.4, while the F1-score decreased by 24.69 points. These findings underline the model's limited capacity to extrapolate from known vulnerabilities to new or rare ones, which poses a significant challenge for real-world applications where unknown vulnerability types frequently arise.

5.6.3 The Ugly

The most critical shortcomings of CodeBERTa's performance were exposed in cross-dataset testing, where the model was trained on one dataset and tested on entirely unseen datasets. This scenario simulates the real-world challenge of deploying a vulnerability detection system trained on one context to operate effectively in a completely different one. The results were stark. When trained on the combined dataset and tested on BigVul, the F1-score plummeted by 69.23 points, dropping from 83.39 to a dismal 14.16. Similarly, for DiverseVul, although some metrics showed minor improvements, key indicators like Recall remained critically low, indicating that the model could not adequately identify vulnerabilities in this setting.

These results point to a deeper issue: the strong influence of dataset-specific biases. Each dataset carries unique characteristics — ranging from labeling strategies to the types of vulnerabilities represented — that the model internalizes during training. While this specificity can boost performance within a dataset, it severely limits generalization across datasets. For instance, while the combined dataset approach improved the model's performance on DiverseVul, it had only marginal benefits—or even detrimental effects—for other datasets like Megavul, where the F1-score saw a slight decline.

This variability underscores the challenges of harmonizing diverse datasets, which often suffer from inconsistencies, noise, and differing definitions of what constitutes

a vulnerability. The poor cross-dataset performance also raises concerns about real-world applicability, where systems must handle diverse and unseen data without the luxury of retraining or fine-tuning on every new context.

TABLE 5.3: CodeBERTa results when tested on unseen data from the three datasets.

Train/Val Dataset	Test Dataset	Previous F1-score	New F1-score	Diff.
Big+Diverse+Mega	BigVul	83.39	83.11	-0.28
Big+Diverse+Mega	DiverseVul	28.5	34.68	+6.18
Big+Diverse+Mega	MegaVul	49.81	49.43	-0.38

5.6.4 Discussion

Overall, the evaluation of CodeBERTa reveals that the model excels in scenarios where the training and testing datasets share similar characteristics, particularly in terms of the same project or similar vulnerability distributions. However, significant challenges arise when faced with unseen projects, vulnerability types, or datasets with varied distributions. The results emphasize the importance of high-quality, diverse datasets for improving the generalization ability of deep learning models in vulnerability detection tasks. Moreover, while combining datasets leads to improved performance, there are still substantial hurdles in ensuring that models can handle the diversity present in real-world data, highlighting the need for further research and improvements in cross-project and cross-vulnerability generalization.

While combining datasets leads to improved performance, the limitations observed in cross-dataset testing highlight the need for more sophisticated data aggregation techniques. Current methods, which primarily focus on concatenation and deduplication, fail to fully address inconsistencies in labeling, representation, and the types of vulnerabilities covered. Techniques such as data augmentation, domain adaptation, and representation learning could help harmonize diverse datasets and reduce dataset-specific biases. Such techniques are critical for developing models capable of performing robustly in real-world applications, where the data are heterogeneous, noisy, and constantly evolving. Future research should focus on integrating these approaches into the data preparation pipeline to enhance the generalization capabilities of transformer-based models in vulnerability detection.

Additionally, employing techniques like hierarchical clustering of vulnerability types or leveraging meta-learning approaches could enable the model to better capture the underlying structure of vulnerabilities, improving its ability to handle unseen CWEs or projects. Another promising direction is the use of active learning, which could prioritize the most informative samples from multiple datasets during training, thereby maximizing the model's exposure to diverse scenarios without introducing excessive noise.

Ultimately, while combining datasets has shown potential, these results underscore the need for advanced aggregation strategies to unlock the full value of diverse datasets. Such techniques are critical for developing models capable of performing robustly in real-world applications, where the data is heterogeneous, noisy, and constantly evolving. Future research should focus on integrating these approaches into the data preparation pipeline to enhance the generalization capabilities of transformer-based models in vulnerability detection.

5.7 Conclusions

This study assessed the impact of dataset quality and diversity on machine learning models for vulnerability detection in source code. By evaluating model performance across unseen projects, new vulnerability types (CWEs), and diverse dataset distributions, we demonstrated that data quality is crucial for the reliability and generalization of these models. The results show that while high-quality datasets enable strong performance in controlled environments, significant declines occur in generalization scenarios, especially when models encounter new projects or vulnerabilities. This highlights the limitations of current datasets, which lack the diversity needed to represent real-world complexity. A key finding is that aggregating datasets enhances model performance. Combining multiple real-world datasets and removing duplicates improved generalization, showing that diverse data distributions mitigate biases. However, simply increasing dataset size is not enough—varied and representative examples are crucial to adapting to different coding styles and vulnerability patterns. Challenges such as imbalanced vulnerability types, inaccurate labels, and limited real-world variability persist. Overcoming these issues requires better dataset curation, improved labeling, and the inclusion of a broader range of vulnerabilities. While synthetic datasets can be useful, they must be designed to reflect real-world complexities. Ultimately, the future of automated vulnerability detection depends more on improving data quality than refining model architectures. Focusing on diverse, accurate, and representative datasets will lead to more reliable and generalizable solutions, advancing software security. These results have been published in the 22nd International Conference on Security and Cryptography (SECRYPT 2025) [8].

Chapter 6

Adversarial attacks to AI-based vulnerability detectors

6.1 Introduction

The growing threat landscape in software security highlights not only the increasing sophistication of real-world attacks but also the vulnerabilities of machine learning–based systems themselves. In particular, models designed for vulnerability detection—while highly effective in many scenarios [26]—are far from immune to adversarial manipulations. As discussed in the introductory chapter, adversarial examples represent a powerful means of testing the robustness of such systems, exposing their reliance on shallow patterns rather than deeper semantic understanding.

In this chapter, we investigated the impact of different adversarial strategies on transformer-based models for vulnerability detection. Our evaluation compared traditional approaches based on syntactic transformations—specifically AST-guided identifier renaming—with more advanced adversarial generation powered by large language models such as ChatGPT. This comparison allowed us to assess not only the relative strength of each attack method but also the underlying weaknesses of current detection models when confronted with perturbations that preserve program semantics.

The results demonstrate a clear distinction between the two strategies. On one hand, the AST-driven approach, rooted in static syntactic structure, introduces moderate disruptions by systematically modifying identifiers while preserving the original control and data flow. On the other hand, ChatGPT-based adversarial samples prove far more damaging: by leveraging natural language–like paraphrasing and diverse transformations, they create adversarial versions of code that remain semantically valid yet consistently evade detection. These findings underline the fundamental limitations of models trained primarily on token-level patterns and reveal the urgent need for representations and defenses that move beyond surface features.

Looking ahead, these insights suggest several promising research directions. One path involves enhancing code representations by incorporating structural and semantic information, for instance through graph-based abstractions that combine syntactic trees, control flow, and data flow. Another lies in hybrid designs where pretrained transformers provide rich contextual embeddings while graph neural networks capture long-range dependencies, potentially increasing robustness against adversarial manipulation. Finally, the development of defense strategies—such as adversarial training with realistic perturbations or detection mechanisms that identify abnormal code transformations—remains a crucial step toward building vulnerability detectors capable of withstanding adversarial pressure in real-world scenarios.

6.2 Background

6.2.1 Adversarial Attacks to Vulnerability Detection models

Recent studies have delved into the security analysis of AI-based vulnerability detectors, uncovering some major weaknesses. Besides their results, in fact, most of the proposed techniques shown severe vulnerability against adversarial attacks. Adversarial attacks are a category of attacks where carefully manipulated inputs can drive a machine learning structure to produce reliably erroneous outputs to an attacker's advantage. The most popular type of attacks to vulnerability detection models is called evasion adversarial attack. With this attack, the attacker subtly modifies input data to cause the machine learning model to misclassify the input, while the changes remain imperceptible to humans. Formally, for a given input sequence $\chi \in X$ where X is the input space, an adversarial code example χ_{adv} needs to satisfy the requirement:

$$\chi \neq \chi_{adv}$$
$$\chi_{adv} \leftarrow \chi + \delta; s.t. ||\delta|| < \theta$$
$$Sim(\chi_{adv}, \chi) \geq \epsilon$$

where θ is the maximum allowed perturbation; $Sim(\cdot)$ is a similarity function; and ϵ is the similarity threshold.

Different adversarial sample generation techniques have been proposed in the literature, and they can be generalized into the following categories:

- **Token replacing:** replacing identifiers or keywords with semantically equivalent alternatives (e.g., variable renaming, function inlining) in order to alter the token distribution without affecting the program semantics. Such transformations are particularly effective against models that heavily rely on lexical or syntactic patterns.
- **Dead code insertion:** injecting non-functional statements (e.g., redundant conditions, unreachable branches, useless assignments) that preserve the original behavior but perturb the input representation. This approach increases the complexity of the code sequence and often misleads models into focusing on irrelevant features.
- **Independent statements switching:** reordering independent instructions or code blocks whose execution order does not affect the program semantics. This strategy disrupts the structural patterns learned by models, while preserving correctness and functionality of the program.

These attack strategies exploit the discrepancy between how models perceive code (mostly as sequences of tokens or graphs) and how compilers or humans interpret it (in terms of functionality and semantics). As a result, they represent a concrete threat to the reliability of AI-based vulnerability detectors.

6.3 Related Works

Recent studies have highlighted the vulnerabilities of machine learning models for software vulnerability detection to adversarial attacks. Liu et al. propose EaTVul

[32], an automatic black-box evasion attack targeting machine-learning-based vulnerability detectors. EaTVul identifies important features in non-vulnerable samples using a combination of Support Vector Machines (SVM) and attention mechanisms. It then uses ChatGPT to generate adversarial code snippets that preserve syntactic correctness and program semantics. The attack operates by inserting dead-code adversarial snippets into vulnerable samples, effectively bypassing detection. Notably, EaTVul achieves a 100% attack success rate in multiple scenarios, demonstrating the fragility of state-of-the-art models such as LineVul and Asteria. Their approach combines prompt optimization and a fuzzy genetic algorithm to select the most effective adversarial samples, showing generalizability across different datasets and even across programming languages like Java. However, EaTVul presents several limitations: it is difficult to replicate due to its reliance on manual steps during the adversarial sample generation phase; additionally, it focuses on Abstract Syntax Tree (AST) structured inputs, which can fail to capture the full semantic richness of source code.

Complementarily, Jha and Reddy introduce CodeAttack [24], a black-box adversarial attack designed to exploit weaknesses in pre-trained programming language models (e.g., CodeT5, CodeBERT, GraphCodeBERT) used for code understanding and generation tasks. CodeAttack targets the "natural channel" of code—variable names, comments, and code structure—applying minimal perturbations such as token replacement guided by masked language models. Unlike previous methods, CodeAttack is agnostic to both programming language and downstream task, demonstrating transferability across tasks such as code translation, repair, and summarization. Empirical results show CodeAttack significantly degrades performance while maintaining high syntactic and semantic similarity with the original code, outperforming traditional NLP attack baselines like TextFooler and BERT-Attack in both attack success rate and imperceptibility. Nonetheless, CodeAttack focuses exclusively on sequence-to-sequence tasks and has not been evaluated on classification tasks, which are prevalent in vulnerability detection settings.

Both EaTVul and CodeAttack share some broader limitations. Their effectiveness is strongly tied to the presence of token patterns in the training data, making them less generalizable in out-of-distribution scenarios. Moreover, both systems exclusively target Transformer-based models and exclude Graph Neural Network (GNN) architectures from their evaluations, leaving a gap in understanding the robustness of graph-based vulnerability detectors.

6.4 Method

In this section, we present the methodology adopted in our study, which is designed to analyze the interaction between adversarial attacks and machine learning models for vulnerability detection. Our approach is structured along two complementary perspectives: the *adversary*, responsible for crafting adversarial samples, and the *victim*, i.e., the machine learning-based vulnerability detector that is exposed to both clean and adversarial data.

6.4.1 Problem Definition

We formalize the problem as follows. Let $x \in X$ be a source code sample with label $y \in \{0, 1\}$ indicating whether it is vulnerable. A vulnerability detector is a function $f_\theta : X \to Y$ parameterized by θ, trained to minimize the prediction error.

An adversary applies a transformation $A(x)$ such that:

$$f_\theta(A(x)) \neq y \quad \text{while preserving the semantics of } x.$$

The defender can attempt to mitigate the adversarial effect, for instance by incorporating adversarial training, where f_θ is trained on a mix of original and adversarial samples.

6.4.2 Victim Perspective

From the defender's side, we consider different model architectures for vulnerability detection, including Transformer-based models and Graph Neural Networks (GNNs). In particular, we evaluate a standard training strategy, where the model is trained exclusively on clean data and subsequently tested against adversarial samples. The objective is to measure the degradation in performance when the victim model is exposed to adversarial inputs.

6.4.3 Adversary Perspective

From the attacker's side, the objective is to craft adversarial samples that are syntactically valid and semantically equivalent to the original code, yet able to mislead the detector. To this end, we adopt the following strategies:

- Replication of the EatVul approach to generate adversarial perturbations;
- Perturbation techniques such as code renaming and dead code insertion;
- Construction of adversarial datasets $D_{adv} = \{A(x) : x \in D\}$ to evaluate the robustness of the victim model.

In the next section we describe the adversarial samples generation process for both of the studied approaches.

6.4.4 Adversarial samples' generation

Adversarial attacks to language models are classified in three categories, based on the granularity of the specific attack: character, token, sentence level. If the simple addition, removal or substitution of a single character may not produce a significant effect on the final result of the model [28], this can not be said about the other two approaches [32]. In this study we explored the effects of two of the most popular families of evasion attacks to Code Language Models (C-LM) for vulnerability detection: vulnerable tokens replacement and not-vulnerable dead code insertion.

Most Relevant Features Extraction

The first step in generating adversarial samples is the identification of the most relevant non-vulnerable features, following the methodology of [32]. Not all code samples contribute equally to a model's prediction; thus, selecting the most informative features is crucial for effective adversarial manipulation. The overall workflow is illustrated in Figure 6.1.

Unlike Liu et al., who relied on SVM and BiLSTM with attention, our implementation employs CodeBERT as the core architecture. The procedure consists of the following steps:

1. **Fine-tuning CodeBERT**: The model is fine-tuned for binary vulnerability detection.
2. **Attention-based feature extraction**: Inference is performed exclusively on non-vulnerable code to obtain attention scores, which are aggregated into a *Token-to-Attention* dictionary.
3. **Substring reconstruction and scoring**: Tokens are de-tokenized to reconstruct substrings, and the attention scores of their constituent tokens are summed to form a *Substring-to-Attention* dictionary.
4. **Selection of relevant substrings**: The top k ($k = 500$ in our experiments) substrings with the highest cumulative attention scores are selected for further analysis.
5. **Keyword extraction using Tree-sitter**: Tree-sitter generates the AST of the code. Nodes identified as `function_declarator` or `call_expression` are labeled as *function*, while other identifiers are labeled as *variable*.
6. **Final filtering**: Symbols and C/C++ keywords are removed to produce the final set of relevant features.

Code Renaming

The first adversarial strategy evaluated in this work is a token renaming attack, which exploits the reliance of Transformer-based models on a limited set of high-impact tokens during prediction. These models assign higher attention weights to the tokens that most strongly influence the final classification decision. Our strategy leverages this behavior to selectively rename critical tokens, aiming to mislead the model while minimizing noticeable changes to the code.

The attack is structured as follows:

- **Definition of Non-Vulnerable Features:**The procedure previously described in 6.4.4. The obtained relevant keywords subset serve as the candidate pool for later token substitutions.
- **Identification of Vulnerability-Critical Tokens:** For each function labeled vulnerable, we use the same model to identify the tokens that received the highest attention weights and contributed most significantly to the vulnerable classification.
- **Token Substitution:** We perform substitutions by replacing vulnerable keywords with alternatives drawn from the non-vulnerable keyword pool. Two approaches are evaluated:
  - **AST-based:** similar to the keyword extraction phase, we leverage Tree-Sitter to extract the AST of the current function, identify the corresponding node types, and substitute them with keywords from the pool.
  - **ChatGPT-based:** in this approach, the extracted important features are used to craft prompts for ChatGPT, explicitly distinguishing between "variable" and "function" identifiers to guide the substitution process.

This strategy aims to maintain the syntactic correctness and semantic plausibility of the modified code, making it difficult for human auditors to detect the changes while still causing the model to misclassify vulnerable code as safe.

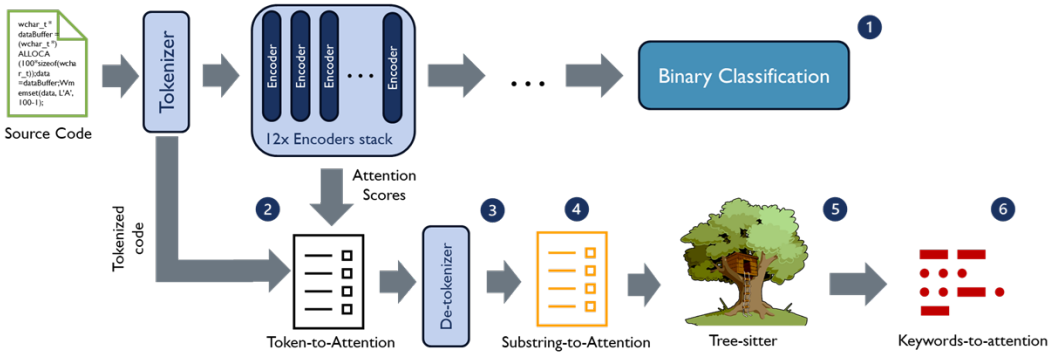


FIGURE 6.1: Most relevant features extraction process

Dead Code Insertion

This adversarial strategy is inspired by EaTVul. Its goal is to transform vulnerable code samples into adversarial versions that are misclassified as non-vulnerable, while preserving the original program semantics. We describe our approach through the following structured steps:

- **Important Feature Extraction**: We use CodeBERT to identify the most influential features within non-vulnerable samples by leveraging the highest attention scores assigned during inference. The input function is first tokenized using CodeBERT's Byte-Pair Encoding (BPE) tokenizer. The resulting tokens are then passed through the 12-layer encoder stack of CodeBERT, which assigns an attention score to each token based on its relevance to the model's final prediction.
- **Important Sample Identification**: Using CodeBERT's tokenizer-specific characters, the token stream is segmented into sublists, each representing a code substring. These substrings are de-tokenized and parsed using Tree-sitter to extract the identifiers. The outcome is a dictionary mapping the most relevant keywords to their corresponding attention scores.
- **Adversarial Snippet Generation (ChatGPT)**: The extracted important features are used to construct prompts for ChatGPT, which generates semantically valid and compilable C code snippets. These snippets are designed to mimic benign behavior while introducing non-functional "dead code" that does not affect program execution.

6.5 Evaluation

To comprehensively assess the effects of the evaluated adversarial attacks to the vulnerability detection models we measure their 'Attack Effectiveness'. Effectiveness denotes the impact of the adversarial samples on the target model's performance. This is evaluated by measuring the degradation in standard classification metrics after injecting adversarial samples into the test set. Specifically, we report: Precision, Recall, and F1-score before and after the attack, quantifying the reduction in model accuracy due to adversarial perturbations. A significant drop in these metrics indicates the success of the attack in compromising the model's predictive capabilities.

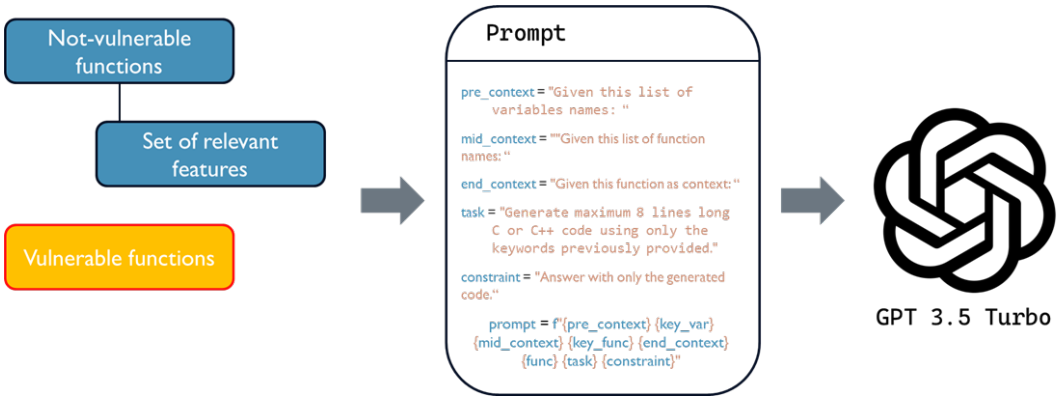


FIGURE 6.2: GPT based adversarial dead code generation process

6.6 Experimental Settings

The experiments were conducted to assess the performance of different models under adversarial data conditions (code renaming, dead code injection), as previously discussed. A GNN model was evaluated, based on the architecture proposed in [21]. Specifically, a Graph Attention Network (GAT) [47] was employed to capture the topological dependency information of source code. In this architecture, source code functions are first represented as Program Dependency Graphs (PDGs), where nodes correspond to statements and edges encode data and control dependencies. Each node is initialized with a contextualized embedding obtained from CodeBERT, which provides token-level semantic representations.

The GAT layers then update node representations by aggregating information from their neighbors through an attention mechanism, which assigns different weights to incoming edges based on their relative importance. Self-loops are added to ensure that a node's own features are preserved during aggregation. By iteratively propagating and refining these representations, the GAT learns to model both syntactic and semantic relationships within the program.

The GNN is evaluated along with several transformer-based architectures: CodeBERT [15], PolyCoder [49], and NatGen [5].

For the graph-based network, the learning rate was set to 1×10^{-5} with AdamW as optimizer. CodeBERT embeddings were employed as input features, and the model was configured for method-level classification only, following the findings of [21]. Training was carried out for 130 epochs.

For the transformer-based models, the learning rate was fixed at 2×10^{-5} with AdamW, using a maximum sequence length of 512 and a batch size of 32. Each model was trained for 10 epochs, with cross-entropy as the loss function.

All trainings were executed on an NVIDIA RTX A6000 GPU.

6.7 Preliminary Results

The results presented in this section provide a preliminary evaluation of the proposed approaches. While they already highlight interesting trends and insights into the effectiveness and limitations of the models, they should be regarded as an initial step in a broader line of research. The next steps for consolidating and extending these findings are discussed in detail in 7.3, where we outline possible directions for further experimentation and improvement.

TABLE 6.1: Performance comparison of all four models on clean data.

Family	Model	Accuracy	Precision	Recall	F1-score
GNN	LineVD	70.19	68.39	66.44	67.40
RoBERTa	CodeBERT	98.11	82.71	85.14	83.91
GPT2	PolyCoder	98.39	87.44	84.31	85.85
T5	NatGen	98.53	86.33	88.62	87.46

6.7.1 GNN Limitations

Table 6.1 highlights a clear performance gap between the graph-based model (LineVD) and the transformer-based architectures. On clean data, LineVD achieves an F1-score of 67.4%, which is substantially lower than the results obtained by CodeBERT (83.91%), PolyCoder (85.85%), and NatGen (87.46%). This discrepancy can be attributed to several limitations intrinsic to GNN-based approaches in this context.

First, while GNNs effectively capture structural and relational information from Code Property Graphs (CPGs), their ability to model long-range dependencies and subtle semantic relations is more constrained compared to transformer-based models, which leverage self-attention mechanisms over the entire input sequence. As a result, GNNs tend to underperform in scenarios where the vulnerability signal is distributed across distant or semantically complex regions of the code.

Second, the reliance on pre-computed embeddings (in this case, CodeBERT embeddings) introduces an additional bottleneck: the graph network is limited by the representational power of the external embeddings, whereas transformers learn contextual representations in an end-to-end manner during training.

Finally, training stability and scalability issues further penalize the GNN model. These results indicate that, although GNNs provide a structured representation of code via CPGs, they struggle to compete with transformer-based architectures in vulnerability detection tasks, even under clean conditions.

6.7.2 Code Renaming Results

To assess the robustness of vulnerability detection models against adversarial perturbations, we evaluated CodeBERT, PolyCoder, and NatGen under two different renaming attack settings. The first setting leverages AST-guided identifier renaming, which firstly identify the candidate identifiers for substitution and then apply statically the adversarial renaming leveraging the relevant features previously extracted. The second employs ChatGPT to generate adversarial variants, resulting in more diverse and less predictable transformations. Tables 6.2 and 6.3 summarize the performance of the three models under both scenarios, reporting precision, recall, and F1-score along with their respective drops compared to clean data.

The comparison between the two adversarial evaluation settings reveals important insights into the robustness of vulnerability detection models under identifier renaming attacks.

In the first scenario, where adversarial samples are generated through AST-guided renaming, performance degradation is noticeable but remains relatively contained. For instance, PolyCoder experiences an F1-score drop of only 4.20, while CodeBERT and NatGen lose 7.61 and 8.37, respectively. This indicates that although AST-based perturbations disrupt lexical cues, they do not fully compromise the models' ability to capture structural or semantic signals. Notably, PolyCoder demonstrates stronger

TABLE 6.2: Performance comparison of CodeBERT, PolyCoder, and NatGen under adversarial code renaming generated via tree-sitter using AST information. Reported values include absolute performance and drops from the clean baseline.

Model	Test Case	Precision	Recall	F1-score
CodeBERT	Clean Data	82.71	85.14	83.91
	Adversarial	80.70	74.40	77.42
	(Drop)	**-2.01**	**-10.74**	**-6.49**
PolyCoder	Clean Data	87.44	84.31	85.85
	Adversarial	86.46	77.34	81.65
	(Drop)	**-0.98**	**-6.97**	**-4.20**
NatGen	Clean Data	86.33	88.62	87.46
	Adversarial	84.58	76.97	80.60
	(Drop)	**-1.75**	**-11.65**	**-6.86**

TABLE 6.3: Performance of CodeBERT, PolyCoder, and NatGen under adversarial code renaming data generated by ChatGPT. Reported values include absolute performance and drops from the clean baseline.

Model	Test Case	Precision	Recall	F1-score
CodeBERT	Clean Data	82.71	85.14	83.91
	Adversarial	74.23	46.98	57.54
	(Drop)	**-8.48**	**-38.16**	**-26.37**
PolyCoder	Clean Data	87.44	84.31	85.85
	Adversarial	80.39	46.82	59.18
	(Drop)	**-7.05**	**-37.49**	**-26.67**
NatGen	Clean Data	86.33	88.62	87.46
	Adversarial	76.15	54.16	63.33
	(Drop)	**-10.18**	**-34.46**	**-24.13**

robustness, suggesting that its architecture is less dependent on surface-level identifiers.

The second scenario, however, paints a more severe picture. When adversarial examples are generated by ChatGPT, the impact on model performance is drastically amplified. CodeBERT's recall collapses from 85.14 to 46.98 (38.16), resulting in a dramatic 26.37 drop in F1-score. PolyCoder, while more resilient in the AST-based case, is equally vulnerable here, with an F1 decline of 26.67. NatGen also suffers substantial degradation, losing 24.13 in F1-score. These results indicate that ChatGPT-generated adversarial samples introduce significantly stronger perturbations, capable of misleading the models into overlooking a large proportion of true vulnerabilities.

A consistent trend across both scenarios is the disproportionate drop in recall compared to precision. This reveals that the models, when attacked, tend to miss vulnerable cases rather than erroneously flagging secure ones. From a security perspective, this is highly problematic, as undetected vulnerabilities pose a far greater risk than false alarms.

Taken together, the results highlight two critical points. First, while AST-based renaming strategies leverage the syntactic structure of the code to identify identifiers

for static modification, their impact on model performance remains relatively moderate. In contrast, ChatGPT-driven adversarial generation, which can create semantically and lexically rich perturbations, produces far more damaging attacks, revealing fundamental limitations in the models' ability to generalize beyond surface-level and syntactic cues. Second, no model demonstrates true robustness across both scenarios: even PolyCoder, which showed resilience under AST-based perturbations, fails to withstand the more sophisticated ChatGPT-generated attacks. This suggests that improving vulnerability detection requires moving beyond token-level robustness and toward architectures and training paradigms that can better encode semantic and structural invariants of source code.

Discussion on ChatGPT-based results

The results reported in Tables 6.2 and 6.3 highlight a clear discrepancy in model degradation between AST-guided renaming and LLM-based renaming. To better understand this gap, we provide a concrete example of how the two approaches transform the original code.

Figure 6.3a shows the original version of the function *__skb_recv_datagram*, while Figure 6.3b illustrates its adversarially perturbed version produced by ChatGPT.

Unlike the AST-guided approach, the LLM-based renaming introduces much richer and less predictable modifications. In the adversarial variant, we observe:

- **Invented identifiers** that do not correspond to the original vocabulary (e.g., *mojom*, *UTF8ToUTF16*, *set_selection_action_data*).
- **Hybrid naming strategies**, where new identifiers emerge as combinations of old names and semantically unrelated terms.
- **Semantic drift**, as the introduced identifiers are often associated with different conceptual domains (e.g., accessibility actions rather than networking).

This behavior results from the generative nature of LLMs, which do not simply replace identifiers but autonomously create new lexical items influenced by training priors. As a consequence, the resulting adversarial code introduces a higher degree of lexical diversity and noise, which significantly challenges the models' reliance on learned token-level and contextual patterns, thus explaining the stronger performance drop observed in the LLM-based setting.

6.7.3 Dead Code Injection Results

The results reported in Table 6.4 illustrate the impact of dead code injection on the performance of CodeBERT, PolyCoder, and NatGen. Overall, the observed drops in precision, recall, and F1-score are significantly smaller compared to adversarial attacks based on identifier renaming, indicating that all three models are relatively robust to this type of perturbation.

Specifically, CodeBERT and NatGen show minimal degradation, with F1-score drops of only 1.28 and 1.12 points, respectively. This suggests that these models are able to effectively ignore irrelevant statements added to the code, likely because their attention mechanisms and contextual embeddings focus on semantically meaningful patterns rather than on superficial syntactic changes.

```
struct sk_buff *__skb_recv_datagram(struct sock *sk, unsigned int flags,
                                    int *peeked, int *off, int *err)
{
        struct sk_buff *skb;
        long timeo;
        int error = sock_error(sk);
        if (error)
                goto no_packet;
        timeo = sock_rcvtimeo(sk, flags & MSG_DONTWAIT);
        do {
                unsigned long cpu_flags;
                struct sk_buff_head *queue = &sk->sk_receive_queue;
                spin_lock_irqsave(&queue->lock, cpu_flags);
                skb_queue_walk(queue, skb) {
                        *peeked = skb->peeked;
                        if (flags & MSG_PEEK) {
                                if (*off >= skb->len) {
                                        *off -= skb->len;
                                        continue;
                                }
                                skb->peeked = 1;
                                atomic_inc(&skb->users);
                        } else
```

(A) Code before ChatGPT adversarial renaming

```
struct mojom *__saved_selection_for_focus_change_(struct ax *mojom, unsigned int
action_data,
                                    int *UTF8ToUTF16, int *set_selection_action_data)
{
        struct mojom *mojom_;
        long kReplaceSelectedText;
        int action = HandleAccessibleAction(mojom);
        if (action)
                goto SelectRange;
        kReplaceSelectedText = UTF8ToUTF16(mojom, action_data & SetUserText);
        do {
                unsigned long action_data_;
                struct mojom_head *action_data_queue = &mojom->mojom_receive_queue;
                spin_lock_irqsave(&action_data_queue->lock, action_data_);
                mojom_queue_walk(action_data_queue, mojom_) {
                        *UTF8ToUTF16 = mojom_->UTF8ToUTF16;
                        if (action_data & Action) {
                                if (*set_selection_action_data >= mojom_->mojom_len) {
                                        *set_selection_action_data -= mojom_->mojom_len;
                                        continue;
                                }
                                mojom_->UTF8ToUTF16 = 1;
                                atomic_inc(&mojom_->users);
                        } else
```

(B) Code after ChatGPT adversarial renaming

FIGURE 6.3: Example of ChatGPT-based adversarial renaming approach on a C++ function.

TABLE 6.4: Performance of CodeBERT, PolyCoder, and NatGen on clean data and adversarial dead code data. Reported values include absolute performance and drops from the clean baseline.

Model	Test Case	Precision	Recall	F1-score
CodeBERT	Clean Data	82.71	85.14	83.91
	Adversarial	82.33	82.94	82.63
	(Drop)	**-0.38**	**-2.20**	**-1.28**
PolyCoder	Clean Data	87.44	84.31	85.85
	Adversarial	86.10	74.59	80.05
	(Drop)	**-1.34**	**-9.72**	**-5.80**
NatGen	Clean Data	86.33	88.62	87.46
	Adversarial	86.05	86.60	86.34
	(Drop)	**-0.28**	**-2.02**	**-1.12**

PolyCoder, in contrast, exhibits a more pronounced decrease, with an F1-score drop of 5.80 points. While still smaller than the drops observed in identifier renaming experiments, this indicates that PolyCoder is somewhat more sensitive to the insertion of dead code. The drop in recall (9.72 points) is particularly noteworthy, suggesting that some vulnerable lines are misclassified due to interference from irrelevant code segments.

These results confirm that dead code injection constitutes a less potent adversarial attack compared to renaming-based attacks, particularly those generated by LLMs, which introduce semantic and lexical changes that directly disrupt the model's learned token-level representations. In practice, this implies that current transformer-based models are largely resilient to trivial syntactic perturbations, but remain highly vulnerable to more sophisticated adversarial manipulations.

6.7.4 Discussion on NatGen results

Among the tested models, NatGen consistently achieved the strongest results, showing both the highest performance on clean data and the greatest robustness under adversarial conditions. Unlike CodeBERT or PolyCoder, NatGen was trained with a naturalizing objective, previously discussed in Chapter 4. During pre-training, the model is exposed to intentionally perturbed but semantically equivalent code, such as variable renaming, dead code injection, or loop restructuring. The learning task requires NatGen to "naturalize" the code back into a human-friendly and canonical form. This continuous exposure to adversarial-like transformations during pre-training makes the model inherently more tolerant to similar perturbations at inference time.

In summary, NatGen's pre-training task provides a form of implicit adversarial training, which explains why it consistently outperforms other transformer-based approaches across different adversarial scenarios. This highlights the importance of designing pre-training objectives that go beyond standard language modeling and directly address the challenges posed by real-world and adversarial code variations.

6.8 Conclusions

The experiments conducted throughout this work provide a comprehensive overview of the robustness and limitations of current AI-based vulnerability detection models

under adversarial scenarios. Across all models—CodeBERT, PolyCoder, and NatGen—we observe that adversarial attacks based on identifier renaming produce the most substantial performance drops, with F1-score reductions exceeding 20 points in some cases, particularly when the adversarial samples are generated by LLMs. This confirms that transformer-based architectures are highly sensitive to lexical and semantic perturbations introduced by generative models, which can create new identifiers or combine existing ones in unpredictable ways.

In contrast, dead code injection appears to be a relatively weak adversarial strategy. Most models, particularly CodeBERT and NatGen, maintain high performance despite the insertion of irrelevant statements, indicating that their learned representations prioritize semantically relevant patterns over superficial syntactic noise. PolyCoder, however, exhibits moderate sensitivity to dead code, suggesting that architectural differences can influence the degree of resilience to such attacks.

These results highlight two main observations: first, the type and generation strategy of adversarial samples strongly influence the model's vulnerability; second, current models are more robust to syntactic perturbations than to semantic or lexical manipulations. Within this landscape, NatGen emerges as the most resilient model, a robustness that can be linked to its naturalizing pre-training objective. This design choice appears to confer an advantage in handling adversarial transformations, reinforcing the importance of pre-training strategies that explicitly account for real-world perturbations.

Chapter 7

Future works

7.1 Deep Learning based Vulnerability Detection

Although the presented studies advance the field of vulnerability detection, several research directions remain open for further exploration. In this section, we outline possible extensions of our work along three complementary lines of research.

7.1.1 Multitask Transformer-based Models

The multitask approach proved effective in simultaneously supporting vulnerability detection and classification, offering security analysts richer information without sacrificing detection accuracy. However, the current experiments were limited to the vulnerability distribution of the BigVul dataset. Future work should explore broader taxonomies of vulnerabilities, testing different grouping strategies for CWE categories and extending the classification task to cover a larger variety of flaws. In addition, designing more sophisticated multitask learning frameworks could enhance the balance between detection and categorization performance, making such models more practical in real-world settings.

7.1.2 Line-level Localization

While the proposed models achieved competitive performance in localizing vulnerabilities at the line level, results indicate that there is still room for improvement compared to specialized systems like LineVul. Future research could investigate hybrid approaches that combine token-level attention mechanisms with richer program representations, such as control-flow or data-flow information. Moreover, integrating explainability techniques could improve the interpretability of the predictions, guiding security analysts more effectively towards the exact location of the vulnerable statements.

7.1.3 LLM-based Approaches

The experiments with Code Llama and Llama 3 highlighted the potential of large language models for vulnerability detection, both at the function and line levels. Future research should investigate fine-tuning strategies specifically tailored to software security tasks, as well as prompt-engineering methods that can better exploit LLM reasoning capabilities. Another promising direction involves designing multi-step pipelines where LLMs are combined with symbolic or static analysis tools, aiming to combine the generative flexibility of LLMs with the rigor of traditional program analysis.

7.2 Data Quality Assessment

Our study has shown that dataset quality and diversity are decisive factors in determining the effectiveness of machine learning models for vulnerability detection. While combining datasets and removing duplicates proved beneficial for generalization, several challenges remain open for future research.

First, future work should focus on improving the accuracy of dataset labels, since many existing resources suffer from weak or inconsistent ground truth. More reliable labeling strategies, potentially combining static and dynamic analysis with expert validation, are required to provide trustworthy benchmarks for training and evaluation. Second, the diversity and representativeness of datasets must be strengthened. Expanding coverage to include a wider range of programming languages, coding styles, project domains, and vulnerability types is essential for building models that can adapt to real-world variability. Current datasets often emphasize specific flaws or contexts, which limits their usefulness for generalization. Finally, synthetic data generation emerges as a promising but delicate avenue. When designed carefully to reflect real-world coding practices and vulnerability patterns, synthetic datasets can complement existing resources by filling gaps and balancing underrepresented categories. A hybrid approach, blending curated real-world samples with realistic synthetic data, could enhance both robustness and generalization. In summary, advancing automated vulnerability detection depends not only on refining model architectures but also, and perhaps more critically, on developing high-quality, diverse, and representative datasets. Addressing these challenges will pave the way for more reliable and widely applicable AI-driven security solutions.

7.3 Adversarial Attacks to AI-based Vulnerability Detection

For future work, several directions are particularly promising. One avenue involves exploring more advanced adversarial sample generation methods that combine both lexical and structural transformations, leveraging the capabilities of LLMs together with program representations such as Code Property Graphs. Another crucial research direction is the development of robust defense mechanisms that can mitigate these attacks. In particular, integrating graph-based information with contextual embeddings from LLMs could enable models to better differentiate between semantically meaningful code and adversarial perturbations, enhancing both vulnerability detection accuracy and resilience. A further promising area lies in investigating specialized pre-training tasks that resemble adversarial training. As demonstrated by NatGen, its robustness can be attributed to a naturalizing pre-training objective, which appears to confer an advantage in handling adversarial transformations. This suggests that pre-training strategies explicitly designed to account for realistic code perturbations could significantly improve model resilience against adversarial attacks. Overall, while current models achieve impressive results on clean data, these experiments underscore the importance of continued research on adversarial robustness and the development of more secure, reliable vulnerability detection systems.

Chapter 8

Conclusions

This thesis has provided a comprehensive investigation into the application of AI, and in particular transformer-based architectures, for the detection and classification of software vulnerabilities. Across multiple studies, we have explored different modeling approaches, assessed the impact of dataset characteristics, and evaluated the robustness of models under adversarial scenarios, providing insights that advance both the theory and practice of automated vulnerability detection.

Our exploration of transformer-based models for function-level vulnerability detection revealed the nuanced influence of dataset composition on model performance. While larger, unbalanced datasets tended to produce better results in vulnerability categorization, balanced datasets improved overall detection accuracy. Furthermore, a multitask approach proved advantageous, offering more informative outputs for security analysts without sacrificing function-level detection accuracy. However, these findings also underscore the limitations of relying on single datasets, as the coverage of vulnerabilities was constrained by the distribution of the BigVul dataset. Extending model capabilities to encompass a wider range of vulnerabilities and exploring alternative grouping strategies remain crucial objectives for future research.

In parallel, the evaluation of Llama 3 and Code Llama for static application security testing (SAST) demonstrated that large language models can achieve state-of-the-art performance in both vulnerability detection and classification. Code Llama, in particular, benefited from its specialized pre-training on source code, resulting in marginally higher accuracy. Importantly, preprocessing strategies such as comment cleaning further enhanced line-level vulnerability localization without degrading performance on downstream tasks, highlighting the value of integrating domain-specific knowledge into preprocessing pipelines.

The study of dataset quality and diversity emphasized that the reliability and generalization of AI-based vulnerability detection models depend more on the characteristics of the training data than on model architecture alone. Models trained on high-quality but homogeneous datasets perform well in controlled settings but experience substantial performance drops when exposed to unseen projects or new vulnerability types. Our findings indicate that combining multiple real-world datasets and removing duplicates improves generalization, mitigating dataset biases and better reflecting the diversity of real-world code. Nevertheless, challenges remain, including imbalanced vulnerability distributions, inaccurate labels, and limited representation of real-world variability. Addressing these issues through careful dataset curation, improved labeling practices, and the design of representative synthetic datasets is essential for advancing the field.

Finally, the analysis of adversarial robustness illuminated critical vulnerabilities of current models. Across all evaluated architectures—CodeBERT, PolyCoder, and

NatGen—lexical perturbations, particularly identifier renaming generated by LLMs, had the most significant impact on performance, whereas dead code injection generally proved less effective. NatGen consistently exhibited superior resilience, an advantage attributable to its naturalizing pre-training objective. These results highlight that pre-training strategies explicitly designed to account for realistic code perturbations can substantially enhance model robustness. Consequently, future research should explore advanced adversarial sample generation techniques, integration of graph-based representations with contextual embeddings, and pre-training paradigms akin to adversarial training to fortify models against lexical and semantic attacks.

In conclusion, this thesis demonstrates that while transformer-based models and LLMs hold significant promise for automated vulnerability detection, their effectiveness is inherently tied to both the quality of the training data and the design of pre-training and defense strategies. Progress in this field will require a holistic approach that combines advanced modeling techniques, high-quality and diverse datasets, and robustness-oriented training paradigms. By addressing these intertwined challenges, the development of reliable, generalizable, and resilient vulnerability detection systems can be realized, ultimately contributing to the security and integrity of modern software systems.

Bibliography

[1] AI@Meta. "Llama 3 Model Card". In: (2024). URL: https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

[2] Marco Ancona et al. *Towards Better Understanding of Gradient-Based Attribution Methods for Deep Neural Networks*. Mar. 7, 2018. DOI: 10.48550/arXiv.1711.06104. arXiv: 1711.06104 [cs, stat]. URL: http://arxiv.org/abs/1711.06104 (visited on 06/15/2023).

[3] Partha Chakraborty et al. "Revisiting the Performance of Deep Learning-Based Vulnerability Detection on Realistic Datasets". In: *IEEE Transactions on Software Engineering* 50.8 (Aug. 2024), pp. 2163–2177. ISSN: 1939-3520. DOI: 10.1109/TSE.2024.3423712. URL: https://ieeexplore.ieee.org/abstract/document/10587162 (visited on 09/03/2024).

[4] Saikat Chakraborty et al. "Deep Learning Based Vulnerability Detection: Are We There Yet?" In: *IEEE Transactions on Software Engineering* 48.9 (Sept. 2022), pp. 3280–3296. ISSN: 1939-3520. DOI: 10.1109/TSE.2021.3087402. URL: https://ieeexplore.ieee.org/abstract/document/9448435 (visited on 09/28/2023).

[5] Saikat Chakraborty et al. "NatGen: Generative Pre-Training by "Naturalizing" Source Code". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE '22: 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Singapore Singapore: ACM, Nov. 7, 2022, pp. 18–30. ISBN: 978-1-4503-9413-0. DOI: 10.1145/3540250.3549162. URL: https://dl.acm.org/doi/10.1145/3540250.3549162 (visited on 11/24/2023).

[6] Yizheng Chen et al. "DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection". In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID 2023: The 26th International Symposium on Research in Attacks, Intrusions and Defenses. Hong Kong China: ACM, Oct. 16, 2023, pp. 654–668. ISBN: 9798400707650. DOI: 10.1145/3607199.3607242. URL: https://dl.acm.org/doi/10.1145/3607199.3607242 (visited on 11/20/2023).

[7] Roland Croft, M. Ali Babar, and M. Mehdi Kholoosi. "Data Quality for Software Vulnerability Datasets". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). May 2023, pp. 121–133. DOI: 10.1109/ICSE48619.2023.00022. URL: https://ieeexplore.ieee.org/abstract/document/10172650 (visited on 11/21/2024).

[8] Claudio Curto, Daniela Giordano, and Daniel Indelicato. "Addressing the C/C++ Vulnerability Datasets Limitation: the Good, the Bad and the Ugly". In: *Proceedings of the 22nd International Conference on Security and Cryptography*. 22nd

International Conference on Security and Cryptography. SCITEPRESS - Science and Technology Publications, 2025, pp. 355–362. ISBN: 978-989-758-760-3. DOI: 10.5220/0013495200003979.

[9] Claudio Curto et al. "Can a Llama Be a Watchdog? Exploring Llama 3 and Code Llama for Static Application Security Testing". In: *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*. 2024 IEEE International Conference on Cyber Security and Resilience (CSR). Sept. 2024, pp. 395–400. DOI: 10.1109/CSR61664.2024.10679444. URL: https://ieeexplore.ieee.org/abstract/document/10679444 (visited on 11/26/2024).

[10] Claudio Curto et al. "MultiVD: A Transformer-based Multitask Approach for Software Vulnerability Detection". In: *Proceedings of the 21st International Conference on Security and Cryptography*. 21st International Conference on Security and Cryptography. Dijon, France: SCITEPRESS - Science and Technology Publications, 2024, pp. 416–423. ISBN: 978-989-758-709-2. DOI: 10.5220/0012719400003767. URL: https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0012719400003767 (visited on 09/11/2024).

[11] Nelson Tavares De Sousa and Wilhelm Hasselbring. "JavaBERT: Training a Transformer-Based Model for the Java Programming Language". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW). Nov. 2021, pp. 90–95. DOI: 10.1109/ASEW52652.2021.00028.

[12] Tim Dettmers et al. "QLoRA: Efficient Finetuning of Quantized LLMs". In: *Advances in Neural Information Processing Systems* 36 (Dec. 15, 2023), pp. 10088–10115. URL: https://proceedings.neurips.cc/paper_files/paper/2023/hash/1feb87871436031bdc0f2beaa62a049b-Abstract-Conference.html (visited on 05/28/2024).

[13] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. May 24, 2019. arXiv: 1810.04805 [cs]. URL: http://arxiv.org/abs/1810.04805 (visited on 02/21/2023). Pre-published.

[14] Jiahao Fan et al. "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries". In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR '20: 17th International Conference on Mining Software Repositories. Seoul Republic of Korea: ACM, June 29, 2020, pp. 508–512. ISBN: 978-1-4503-7517-7. DOI: 10.1145/3379597.3387501. URL: https://dl.acm.org/doi/10.1145/3379597.3387501 (visited on 02/21/2023).

[15] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. Sept. 18, 2020. DOI: 10.48550/arXiv.2002.08155. arXiv: 2002.08155 [cs]. URL: http://arxiv.org/abs/2002.08155 (visited on 03/06/2023). Pre-published.

[16] Michael Fu and Chakkrit Tantithamthavorn. "LineVul: A Transformer-based Line-Level Vulnerability Prediction". In: (2022). DOI: 10.1145/3524842. URL: https://doi.org/10.1145/3524842..

[17] Michael Fu et al. *ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We?* Oct. 15, 2023. arXiv: 2310.09810 [cs]. URL: http://arxiv.org/abs/2310.09810 (visited on 02/28/2024). Pre-published.

[18] Michael Fu et al. "VulExplainer: A Transformer-based Hierarchical Distillation for Explaining Vulnerability Types". In: *IEEE Transactions on Software Engineering* (2023), pp. 1–17. ISSN: 1939-3520. DOI: 10.1109/TSE.2023.3305244.

[19] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. "Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey". In: *ACM Computing Surveys* 50.4 (Aug. 1, 2017). ISSN: 15577341. DOI: 10.1145/3092566.

[20] Daya Guo et al. "GraphCodeBERT: Pre-training Code Representations with Data Flow". In: International Conference on Learning Representations. Oct. 2, 2020. URL: https://openreview.net/forum?id=jLoC4ez43PZ (visited on 02/16/2024).

[21] David Hin et al. "LineVD: Statement-Level Vulnerability Detection Using Graph Neural Networks". In: *Proceedings of the 19th International Conference on Mining Software Repositories*. MSR '22: 19th International Conference on Mining Software Repositories. Pittsburgh Pennsylvania: ACM, May 23, 2022, pp. 596–607. ISBN: 978-1-4503-9303-4. DOI: 10.1145/3524842.3527949. URL: https://dl.acm.org/doi/10.1145/3524842.3527949 (visited on 01/08/2024).

[22] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. Oct. 16, 2021. arXiv: 2106.09685 [cs]. URL: http://arxiv.org/abs/2106.09685 (visited on 12/04/2023). Pre-published.

[23] Hamel Husain et al. *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search*. June 8, 2020. arXiv: 1909.09436 [cs, stat]. URL: http://arxiv.org/abs/1909.09436 (visited on 02/14/2023). Pre-published.

[24] Akshita Jha and Chandan K. Reddy. *CodeAttack: Code-Based Adversarial Attacks for Pre-trained Programming Language Models | Proceedings of the AAAI Conference on Artificial Intelligence*. URL: https://ojs.aaai.org/index.php/AAAI/article/view/26739 (visited on 02/18/2025).

[25] Xuefeng Jiang et al. *Investigating Large Language Models for Code Vulnerability Detection: An Experimental Study*. Jan. 5, 2025. DOI: 10.48550/arXiv.2412.18260. arXiv: 2412.18260 [cs]. URL: http://arxiv.org/abs/2412.18260 (visited on 01/07/2025). Pre-published.

[26] Ilias Kalouptsoglou et al. "Software Vulnerability Prediction: A Systematic Mapping Study". In: *Information and Software Technology* 164 (Dec. 2023), p. 107303. ISSN: 09505849. DOI: 10.1016/j.infsof.2023.107303. URL: https://linkinghub.elsevier.com/retrieve/pii/S095058492300157X (visited on 01/16/2024).

[27] Katikapalli Subramanyam Kalyan, Ajit Rajasekharan, and Sivanesan Sangeetha. *AMMUS : A Survey of Transformer-based Pretrained Models in Natural Language Processing*. Aug. 28, 2021. arXiv: 2108.05542 [cs]. URL: http://arxiv.org/abs/2108.05542 (visited on 10/05/2023). Pre-published.

[28] Marina Katoh, Weiping Pei, and Youye Xie. "AdVul: Adversarial Attack against ML-based Vulnerability Detection". In: *2024 Annual Computer Security Applications Conference Workshops (ACSAC Workshops)*. 2024 Annual Computer Security Applications Conference Workshops (ACSAC Workshops). Dec. 2024, pp. 97–107. DOI: 10.1109/ACSACW65225.2024.00018. URL: https://ieeexplore.ieee.org/document/10917677/ (visited on 05/02/2025).

[29] Yi Li, Shaohua Wang, and Tien N. Nguyen. "Vulnerability Detection with Fine-Grained Interpretations". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 18, 2021, pp. 292–303. ISBN: 978-1-4503-8562-6. DOI: 10.1145/3468264.3468597. URL: https://dl.acm.org/doi/10.1145/3468264.3468597 (visited on 06/15/2023).

[30] Chongyang Liu et al. "Making Vulnerability Prediction More Practical: Prediction, Categorization, and Localization". In: *Information and Software Technology* (Mar. 28, 2024), p. 107458. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2024.107458. URL: https://www.sciencedirect.com/science/article/pii/S0950584924000636 (visited on 04/03/2024).

[31] Ruitong Liu et al. *Source Code Vulnerability Detection: Combining Code Language Models and Code Property Graphs*. Version 1. Apr. 23, 2024. DOI: 10.48550/arXiv.2404.14719. arXiv: 2404.14719 [cs]. URL: http://arxiv.org/abs/2404.14719 (visited on 06/17/2025). Pre-published.

[32] Shigang Liu et al. "EaTVul: ChatGPT-based Evasion Attack Against Software Vulnerability Detection | USENIX". In: 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/liu-shigang.

[33] Scott M Lundberg and Su-In Lee. "A Unified Approach to Interpreting Model Predictions". In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/8a20a8621978632d76c43dfd28b67767-Abstract.html (visited on 06/15/2023).

[34] Cláudia Mamede et al. "Exploring Transformers for Multi-Label Classification of Java Vulnerabilities". In: *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS). Dec. 2022, pp. 43–52. DOI: 10.1109/QRS57517.2022.00015.

[35] Sourab Mangrulkar et al. *PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods*. https://github.com/huggingface/peft. 2022.

[36] MITRE. *CVE published by year*. https://www.cvedetails.com/browse-by-date.php. Accessed: 2024-02-16. Feb. 16, 2024.

[37] National Institute of Standards and Technology. *NIST software assurance reference dataset*. https://samate.nist.gov/SARD. Accessed: 2025-01. 2025.

[38] Chao Ni et al. "MegaVul: A C/C++ Vulnerability Dataset with Comprehensive Code Representations". In: *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. 2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR). Apr. 2024, pp. 738–742. URL: https://ieeexplore.ieee.org/document/10555623/?arnumber=10555623 (visited on 09/02/2024).

[39] Colin Raffel et al. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". In: *Journal of Machine Learning Research* 21.140 (2020), pp. 1–67. ISSN: 1533-7928. URL: http://jmlr.org/papers/v21/20-074.html (visited on 05/28/2024).

[40] Baptiste Rozière et al. "Code Llama: Open Foundation Models for Code". In: ().

[41] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. "Learning Important Features Through Propagating Activation Differences". In: *Proceedings of the 34th International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, July 17, 2017, pp. 3145–3153. URL: https://proceedings.mlr.press/v70/shrikumar17a.html (visited on 06/15/2023).

[42] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. *Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps*. Apr. 19, 2014. DOI: 10.48550/arXiv.1312.6034. arXiv: 1312.6034 [cs]. URL: http://arxiv.org/abs/1312.6034 (visited on 06/15/2023). Pre-published.

[43] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. "Axiomatic Attribution for Deep Networks". In: *Proceedings of the 34th International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, July 17, 2017, pp. 3319–3328. URL: https://proceedings.mlr.press/v70/sundararajan17a.html (visited on 06/15/2023).

[44] Zhenzhou Tian et al. "Enhancing Vulnerability Detection via AST Decomposition and Neural Sub-Tree Encoding". In: *Expert Systems with Applications* 238 (Mar. 2024), p. 121865. ISSN: 09574174. DOI: 10.1016/j.eswa.2023.121865. URL: https://linkinghub.elsevier.com/retrieve/pii/S0957417423023679 (visited on 04/20/2025).

[45] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. July 19, 2023. arXiv: 2307.09288 [cs]. URL: http://arxiv.org/abs/2307.09288 (visited on 12/07/2023). Pre-published.

[46] Ashish Vaswani et al. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html (visited on 07/25/2023).

[47] Petar Veličković et al. *Graph Attention Networks*. Feb. 4, 2018. DOI: 10.48550/arXiv.1710.10903. arXiv: 1710.10903 [stat]. URL: http://arxiv.org/abs/1710.10903. Pre-published.

[48] Yue Wang et al. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. Sept. 2, 2021. DOI: 10.48550/arXiv.2109.00859. arXiv: 2109.00859 [cs]. URL: http://arxiv.org/abs/2109.00859 (visited on 11/24/2023). Pre-published.

[49] Frank F. Xu et al. "A Systematic Evaluation of Large Language Models of Code". In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. MAPS 2022. New York, NY, USA: Association for Computing Machinery, June 13, 2022, pp. 1–10. ISBN: 978-1-4503-9273-0. DOI: 10.1145/3520312.3534862. URL: https://dl.acm.org/doi/10.1145/3520312.3534862 (visited on 05/22/2024).

[50] Jian Zhang et al. *An Empirical Study of Automated Vulnerability Localization with Large Language Models*. Mar. 30, 2024. arXiv: 2404.00287 [cs]. URL: http://arxiv.org/abs/2404.00287 (visited on 05/21/2024). Pre-published.

[51] Yu Zhang and Qiang Yang. "An Overview of Multi-Task Learning". In: *National Science Review* 5.1 (Jan. 1, 2018), pp. 30–43. ISSN: 2095-5138, 2053-714X. DOI: 10.1093/nsr/nwx105. URL: https://academic.oup.com/nsr/article/5/1/30/4101432 (visited on 10/05/2023).

[52] Yunhui Zheng et al. "D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). May 2021, pp. 111–120. DOI: 10.1109/ICSE-SEIP52600.2021.00020. URL: https://ieeexplore.ieee.org/abstract/document/9402126 (visited on 11/26/2024).

[53] Yaqin Zhou et al. "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html (visited on 10/14/2024).

[54] Deqing Zou et al. "µVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection". In: *IEEE Transactions on Dependable and Secure Computing* 18.5 (Sept. 23, 2019), pp. 2224–2236. ISSN: 1941-0018. DOI: 10.1109/TDSC.2019.2942930.