Information Retrieval-Driven Software Vulnerability Prediction

Chizzy Godson Meka

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

April 14, 2025

I, Chizzy Godson Meka, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

Context: The growing complexity of software systems continually correlates with increasing vulnerabilities, necessitating effective mitigation strategies, such as vulnerability prediction. This Artificial Intelligence (AI)-driven approach aims to improve Secure Software Development Life Cycle (SSDLC) practices by proactively identifying potential security flaws. While various prediction approaches have been proposed, opportunities for further research remain, particularly in leveraging Information Retrieval (IR)'s pattern-matching capabilities to enhance prediction models.

Objective: This thesis advances secure software engineering methodologies by introducing IR-driven feature engineering methods for predicting software vulnerabilities. We develop granular method-level vulnerability prediction models that leverage novel IR-driven security-relevant metrics and evaluate their predictive performances.

Methodology: We developed two varieties of sixteen IR-driven security-relevant features using token-based and Abstract Syntax Tree (AST)-based source code representations. Then, we utilised these features to develop models with various machine learning classifiers using Python and evaluated them on Java open-source software systems, starting with a Within-Project (release-by-release dataset). Finally, we conducted a stress test in a Mixed-Project (multi-software systems dataset) setting to assess the generalisability of our models across software systems.

Results: Our Within-Project token-based IR-driven approach reached a post-hyperparameter tuning precision of *0.73*, a recall of *0.60*, and an F1 score of *0.66* using a Random Forest classifier. The Within-Project AST-based approach attained

a slightly better F1 score performance, yielding a post-hyperparameter tuning precision of *0.72*, a recall of *0.62*, and an F1 score of *0.67*, also using Random Forest.
Conclusion: Our research indicates that IR-driven feature engineering techniques significantly enhance prediction performance, demonstrating the effectiveness of our approach. However, the Mixed-Project analysis indicated that data-related challenges in vulnerability prediction persist, especially regarding data heterogeneity across software systems. Thus, system-specific vulnerability prediction models leveraging a release-by-release dataset and knowledge of previous system-specific vulnerabilities represent the most promising approach for practical vulnerability prediction in real-world software systems.

Impact Statement

The research presented in this thesis has the potential to make significant contributions both within academia and beyond. In the academic sphere, this work advances the software engineering field, particularly in software vulnerability prediction. By integrating information retrieval techniques into vulnerability prediction models, the research introduces a novel approach that could reshape how vulnerabilities are detected in software systems. This could lead to more accurate and scalable prediction models, which are crucial given the increasing complexity and size of modern software systems.

Beyond academia, this research could profoundly impact on several industries, particularly those that rely heavily on software systems. Developing more robust software vulnerability prediction models could enhance security measures, reducing the likelihood of successful cyberattacks. This, in turn, could help maintain public trust in digital systems by preventing reputational damage by protecting sensitive data. Such advancements could have far-reaching implications for industries such as finance, healthcare, and critical infrastructure, where software security is paramount. They could help safeguard against more severe potential catastrophic consequences, such as financial losses, injury, or loss of life in critical systems.

Moreover, the insights gained from this research could inform the development of new tools and practices for software developers. By incorporating information retrieval-driven techniques, software professionals could be better equipped to identify and mitigate vulnerabilities during the software development lifecycle. Such development could help reduce the Time To Market (TTM) for software products

while improving their security, benefiting end-users by providing safer and more reliable digital environments.

The research could also influence public policy, particularly in cybersecurity-related areas. As governments and organisations worldwide seek to strengthen their defences against cyber threats, the methodologies and findings presented in this thesis could serve as a foundation for new policies and regulations to improve software security standards. This could have a wide-reaching impact, not only on the organisational level but also on national and international cybersecurity strategies.

In summary, this thesis contributes to both the theoretical advancement of software vulnerability prediction and its practical application in the real world. The potential benefits of this work are far-reaching, with implications for academic research, industry practices, and public policy. This research has the potential to impact how vulnerabilities are managed in the digital age by addressing the pressing issue of software security through innovative methods.

Acknowledgements

The high-quality supervision I received from Dr. Jens Krinke is a testament to his dedication. This attribute is profound enough to be immediately apparent upon interacting with him. His support and guidance as my primary supervisor remained unwavering from the time I expressed my intention to enrol in the PhD programme in the Computer Science department until I finalised this thesis, and I will be forever thankful. I am also grateful to my second supervisor, Dr. Ingolf Becker, for his support in the earlier part of my doctoral journey.

Beyond UCL, my spouse and two boys have most positively impacted my life on this academic journey, and I owe them the privilege of being in a position to write these acknowledgements. My wife does not know a lot about software vulnerability prediction. Still, the cumulative effect of her immeasurable indirect support has compounded into a positive force that has significantly contributed to the realisation of this thesis; even better, as the academically minded individual that she is, my journey has sparked her interest in pursuing her doctoral studies, and I am here for it. This positive influence is what I also look forward to my two boys partaking in. I am blessed that our sons are young enough to have only known me as a doctoral student and that their earliest memories of me would be witnessing their father living that "PhD life." That sentiment contributed to my determination to see this programme through and set an example about setting and achieving life goals.

I also acknowledge my wider family and friends and eagerly look forward to reciprocating the favour.

I am thankful for overcoming the challenges to reach this point, proud to achieve this milestone, and eternally grateful for this academic experience.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This chapter outlines the research context of this thesis, concentrating on software bugs and vulnerabilities, their impact, and the associated challenges. It further details the motivation, research questions, scope, contributions, and structure of the thesis.

1.1 Software Bugs

Computer systems are the foundation of our digital society, empowering individuals and companies globally [Lin et al., 2020b]. However, their ubiquity brings the proliferation of software bugs, a primary concern for software professionals and users alike. A software bug is an error, flaw, or fault in a program that leads to incorrect or unexpected outcomes or unpredictable behaviour. Bugs can range from minor issues to serious system failures with potentially devastating consequences. Most bugs stem from human errors in the source code or design of a software system [Singh, 2013]. These mistakes can occur during coding, testing, or maintenance. Since the first documented software bug in 1945, professionals have been developing techniques to mitigate these issues [Sophia et al., 2021]. Despite these efforts, software bugs continue to be a significant concern.

Bugs are costly to resolve and increase maintenance efforts [Sophia et al., 2021]. The cost of fixing bugs escalates exponentially as the software development lifecycle progresses. Additionally, they directly impact a software system's quality [Khan et al., 2020], leading to user inconvenience and software crashes.

Bugs can be classified based on their impact on a system's functionality. Non-functional bugs affect a system's non-functional requirements. They may have a minor impact on functionality and remain undetected but can cause crashes or freezes, leading to denial of service. In contrast, functional bugs affect a system's functional requirements, and performance-related bugs impair a system's performance [Sophia et al., 2021]. Finally, there are security-related bugs, also known as vulnerabilities.

Our research focuses on vulnerabilities. These bugs pose a significant threat in modern society, where information systems support most daily processes. For example, they can enable unauthorised access to a software system.

1.2 Software Vulnerabilities

Software vulnerabilities, like other bugs, could arise from design, development, or configuration flaws. However, unlike other bugs, malicious actors can exploit these flaws to violate security policies [Lin et al., 2020b]. A software vulnerability is a weakness in a system that can be exploited by threats, adversely impacting the confidentiality, integrity, and availability of the affected systems [Spanos and Angelis, 2018].

Vulnerabilities differ from general bugs as they represent security-related issues, while traditional bugs indicate impaired or insufficient functionality [Camilo et al., 2015]. Not all software weaknesses are exploitable or available for abuse, so not all qualify as vulnerabilities.

Software vulnerabilities are a significant concern for professionals. Despite countermeasures, software attacks cost up to $200 billion per year. The Common Vulnerabilities and Exposures (CVE) system reports that the number of vulnerabilities has more than doubled in recent years, heightening security concerns due to the potential for malicious exploitation [Gupta et al., 2021].

1.2.1 How Software Vulnerabilities Arise

Software system vulnerabilities can result from various factors, including coding errors, design flaws, inadequate input validation, and improper system configuration. Each phase of the software development lifecycle can introduce vulnerabilities.

Coding errors are a significant source of vulnerabilities. Programmers may inadvertently introduce bugs and logic errors that attackers can exploit, known as vulnerability-inducing changes. For example, an input validation flaw in a web application may allow an attacker to inject malicious code, or a buffer overflow can enable arbitrary code execution [Gujral et al., 2015].

Design flaws can also introduce vulnerabilities. Poor system design can make it easier for attackers to exploit weaknesses [Igure and Williams, 2008]. For instance, storing passwords in plain text instead of encrypting them allows an attacker who gains access to the database to steal all the passwords. Similarly, weak authentication mechanisms, such as simple passwords, make it easy for an attacker to guess or crack them, thereby gaining access to the system [Nandy et al., 2019].

Lack of input validation is another common source of vulnerabilities. Unvalidated user input can allow attackers to inject malicious code into the system. For example, SQL injection attacks occur when an attacker uses unvalidated input to modify or extract data from a database [Fadlalla and Elshoush, 2023].

Poor system configuration can lead to vulnerabilities. Misconfigured systems can have open ports, unsecured databases, or other weaknesses that attackers can exploit. For example, leaving the default password on a database or server makes it easy for an attacker to access the system, as default passwords are often easily found in documentation or online sources [OWASP, 2021].

Human error is often a significant factor in creating vulnerabilities [Pollock, 2017]. To address these errors, it is essential to identify and proactively mitigate vulnerabilities. Changes made to fix these vulnerabilities are referred to as vulnerability-fixing changes [Bhandari et al., 2021].

1.2.2 The Ever-Present Threat of Software Vulnerabilities

In 2013, Apple's Developer portal suffered an information theft breach[1]. In 2018, British Airways faced a £183 million penalty after a data breach compromised 380,000 customers' transaction details[2] [3]. In 2020, the US government suffered multiple data breaches at top federal agencies, attributed to a cyberattack suspected to have been orchestrated by Russia[4]. In 2021, Microsoft reported an exploit by a hacker group allegedly working for China, targeting Microsoft Exchange servers via zero-day vulnerabilities[5]. In mid-2023, the Superannuation Arrangements of the University of London (SAUL) experienced a data breach due to a cyber incident

[1]https://www.theguardian.com/technology/2013/jul/22/apple-developer-site-hacked

[2]https://www.reuters.com/article/us-iag-cybercrime-british-airways/ba-apologizes-after-380000-customers-hit-in-cyber-attack-idUSKCN1LM2P6

[3]https://www.bbc.co.uk/news/business-48905907

[4]https://edition.cnn.com/2020/12/16/tech/solarwinds-orion-hack-explained/index.html

[5]https://www.nbcnews.com/tech/security/u-s-issues-warning-after-microsoft-says-china-hacked-its-n1259522

involving MOVEit software[6]. In early 2023, T-Mobile discovered an API vulnerability that led to the theft of personal data belonging to 37 million customers[7]. In June 2024, a ransomware attack targeted NHS hospitals, disrupting medical services and necessitating an urgent appeal for blood donations[8] [9].

These examples illustrate the indiscriminate nature of cyberattacks, affecting governmental, commercial, non-profit, and individual entities. Exploitation of vulnerabilities is a common theme in all these cases, highlighting the severe consequences of neglecting software system vulnerabilities. Consequences range from minor inconveniences, such as denial of service, to significant issues, including reputational damage, financial loss, or even injury and death in safety-critical systems. Therefore, it is crucial to identify and mitigate vulnerabilities before they can be exploited by attackers.

Despite the importance of identifying and eradicating vulnerabilities, developers must balance the cost of remediation, the potential impact of the vulnerability, and the overall software development lifecycle. An automated process that effortlessly identifies and prioritises security weaknesses can be invaluable. Such a solution can help developers to focus on the most critical vulnerabilities, reducing remediation time and cost while ensuring a productive software development lifecycle.

[6]`https://www.ucl.ac.uk/human-resources/news/2023/jun/saul-response-cyber-incident-and-data-breach`

[7]`https://techcrunch.com/2023/01/19/t-mobile-data-breach/`

[8]`https://news.sky.com/story/nhs-issues-urgent-blood-donation-appeal-after-it-cyber-attack-leaves-hospitals-struggling-to-match-patients-13150509`

[9]`https://news.sky.com/story/nhs-cyber-attack-sensitive-data-stolen-from-blood-test-provider-in-cyber-attack-by-criminal-group-published-online-13154539`

1.3 Motivation

Our research is driven by the increasing threat of software vulnerabilities and the urgent need for effective vulnerability prediction methods. In recent years, the National Vulnerability Database (NVD)[10] has reported a steady increase in disclosed security vulnerabilities. Software vulnerabilities can have severe consequences if exploited, making it essential to identify and mitigate them proactively. Traditional vulnerability prediction methods, such as static code analysis and dynamic code analysis, face limitations, particularly in large-scale software systems, due to high computational costs [Shin and Williams, 2013]. Furthermore, these methods rely heavily on manual feature definition and code audits, which are both time-consuming and prone to errors [Kaur and Nayyar, 2020, Lipp et al., 2022]. The increasing complexity and variety of software systems further complicate vulnerability identification using these conventional techniques [Zhang et al., 2023c]. As a result, researchers are now focusing on automated, AI-driven methods, using machine learning and deep learning to enhance the efficiency and accuracy of vulnerability prediction.

However, while machine learning techniques offer promise, they face challenges that impact their performance. A fundamental limitation in existing machine learning-driven vulnerability prediction models is the quality of features used for training. Most models rely on features extracted from source code, such as metrics of complexity and structure. However, these features often lack security context, which is critical for capturing the full implications of the code [Lin et al., 2020a]. For example, a complex code snippet might not be vulnerable if inaccessible to an attacker, yet traditional features may not capture this information. Therefore, incorporating security context into feature extraction is vital for improving vulnerability prediction. High false positive rates are another significant issue with current machine learning models [Shin and Williams, 2013]. False positives, where non-vulnerable code is incorrectly flagged as vulnerable, waste time and resources. Reducing false positives is crucial to making these models more effective in practice.

[10] https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time

To address these limitations—namely, the lack of security context in features and high false positive rates—we propose a novel approach that utilises supervised machine learning techniques for vulnerability prediction, enhanced by information retrieval methods.

Information retrieval, widely used in search engines and text mining, involves retrieving relevant information from extensive data collections [Chowdhury, 2010]. Our research adapts this concept for vulnerability prediction, aiming to improve the quality of training features and reduce false positives. The core of information retrieval is pattern matching. Given a query, an information retrieval system retrieves relevant documents from a collection based on the query's similarity to the documents. In our context, the 'query' is a code element from a software system being analysed for vulnerabilities, and the 'documents' are known vulnerable code samples from a dataset. Our system retrieves relevant documents based on their similarity to the query, extracting features that encapsulate the security context of the code element. We then use these features to train machine learning models.

By 'code element', we refer to a unit of analysis in the source code, also known as 'granularity.' In modern object-oriented programming languages like Java, this could be a method or a class. For instance, in method-scoped analysis, the code element is a method, and the information retrieval system retrieves relevant documents, i.e., vulnerable methods, based on their similarity to the method under analysis. Similarly, in class-scoped analysis, the code element is a class, and the system would retrieve relevant documents, i.e., vulnerable classes, based on their similarity to the class under analysis. Our research focuses on method-level granularity, predicting whether a method is vulnerable based on its source code. Consequently, the resulting features are method-level.

By matching queries to documents, we extract quantitative attributes that capture the security context of the methods we analyse relative to the known vulnerable methods in the vulnerability dataset. These attributes, derived using novel similarity metrics, serve as features for training machine learning models. This process embeds the security posture of the methods we analyse into the features relative to the comprehensiveness of the vulnerability dataset, enhancing the model's ability to predict vulnerabilities in unseen, similar code.

In summary, our research introduces a novel approach to vulnerability prediction by integrating information retrieval techniques and a dataset of known vulnerable code samples. This approach integrates security context into the training features, aiming to reduce false positives and ultimately enhance the performance of machine learning models in vulnerability prediction.

By addressing these limitations, our research aims to provide software developers with a more practical and effective method for identifying vulnerability-prone components in software systems before they can be exploited.

1.4 Research Questions

The research questions guiding this thesis are as follows:

1. How well does the information retrieval-driven software vulnerability prediction technique perform on a single, multi-release software system dataset for token-based source code representations?

2. How well does the information retrieval-driven software vulnerability prediction technique perform on a single, multi-release software system dataset for Abstract Syntax Tree (AST)-based source code representations?

3. How well does the information retrieval-driven software vulnerability prediction technique generalise across multiple software systems?

The first and second research questions evaluate the performance of our information retrieval-driven software vulnerability prediction technique on a single, multi-release software system dataset using different source code representations. These evaluations will offer insights into the technique's effectiveness in predicting vulnerabilities.

The third research question examines the generalisation capabilities of the information retrieval-driven software vulnerability prediction technique across multiple software systems, as a form of stress test, providing insights into its ability to predict vulnerabilities in within-project and mixed-project datasets. A 'within-project' dataset contains data from a single software system, whereas a 'mixed-project' dataset contains data from multiple software systems.

1.5 Research Scope

1. This research focuses on predicting software vulnerabilities using supervised machine learning techniques.

2. The study develops a novel approach leveraging information retrieval techniques and a dataset of known vulnerable code samples to extract features and train machine learning models. Other machine learning techniques, such as unsupervised machine learning, reinforcement learning, and deep learning, are not considered.

3. The study evaluates the performance of the approach on Java-based software systems.

4. The unit of analysis is the method in the source code of software systems.

5. Only code-related vulnerabilities occurring within methods are considered. We do not consider configuration-related, design-related, or other types of vulnerabilities.

6. The study focuses on binary classification tasks to predict whether a method is vulnerable rather than considering multi-class classification tasks, which focus on predicting the type of vulnerability.

1.6 Research Contributions

Information retrieval is a well-established technique for extracting relevant information from large text corpora. However, its application to software vulnerability prediction has been largely unexplored. This research demonstrates the feasibility of repurposing information retrieval techniques for practical vulnerability prediction. By harnessing the pattern-matching capabilities of information retrieval, we develop novel security-related software metrics (features) that facilitate encoding security context into machine learning features to enhance the effectiveness of machine learning models in identifying vulnerable code.

The primary contributions of this research are as follows:

1. A novel approach to software vulnerability prediction that applies information retrieval techniques to extract security-relevant features from source code by leveraging known vulnerable code samples.

2. A comprehensive evaluation of this approach using a multi-release dataset of a single software system, incorporating both token-based and AST-based representations of source code.

3. An assessment of the generalisability of this approach across multiple software systems, offering insights into its performance on both within-project and mixed-project datasets.

4. A comparative analysis of this approach against existing machine learning-based software vulnerability prediction models.

5. A discussion of the research findings' implications for software developers and the broader software security community, providing insights into its practical applications and potential impact on vulnerability prediction.

6. A set of recommendations for future research in software vulnerability prediction, outlining potential areas for further investigation and development.

1.7 Thesis Outline

The rest of this thesis is structured as follows:

- Chapter 2 provides background information on the concepts and techniques used in our research.
- Chapter 3 reviews the literature on software vulnerability prediction and related topics.
- Chapter 4 presents a study on information retrieval-driven software vulnerability prediction using token-based representations.
- Chapter 5 presents a study on information retrieval-driven software vulnerability prediction using AST-based representations (Code2Vec).
- Chapter 6 examines the generalisability of information retrieval-driven software vulnerability prediction across multiple software systems.
- Chapter 7 concludes the thesis, summarising the research contributions and discussing future work.
- Appendix A investigates the co-evolution of bug-fixing change and bug-inducing change artefacts.
- Appendix B explores using Large Language Models for vulnerability prediction.
- Finally, the thesis features a comprehensive glossary of the most relevant terms defined in the context of this research. Each term in the glossary features a page number reference to the first occurrence of the term in the thesis. Note that terms that appear in the Literature Review chapter or any other parts of the thesis discussing other scholars' work do not qualify as first occurrences to avoid misdefining them in ways that are not aligned with the original authors' definitions.

Chapter 2

Background

This chapter covers the background of software vulnerabilities, mitigation techniques, vulnerability prediction, and representations of source code. We explore the granularity and features of vulnerability prediction models, as well as the source code representations employed in our research. Additionally, we introduce information retrieval concepts and their relevance to vulnerability prediction in our research.

2.1 Software Vulnerability Mitigation Techniques

Attacks exploiting software vulnerabilities can take various forms, such as Denial-of-Service (DoS) attacks or privilege escalation. One trusted mitigation method is adhering to secure programming practices during development. This reduces the risk of introducing vulnerabilities and simplifies their management. Despite efforts, including Microsoft reportedly spending 100 machine years annually addressing software flaws [Chernis and Verma, 2018], many flaws still reach production.

Researchers have proposed various methods to tackle vulnerabilities, including static analysis, dynamic analysis, hybrid analysis, penetration testing, patching, and program transformation. Recently, AI-driven methodologies, primarily machine learning and deep learning approaches, have been the focus [Ghaffarian and Shahriari, 2017]. Before these machine learning-driven techniques, static and dynamic analysis techniques were standard. Static analysis examines source code without executing it to identify potential vulnerabilities, analysing code structure, data flow, and control flow [Ghaffarian and Shahriari, 2017]. Techniques include code metrics analysis, pattern matching, and symbolic execution [Zhou et al., 2021, Chen et al., 2017, Luckow et al., 2020]. Dynamic analysis involves executing code to observe its behaviour and identify vulnerabilities during runtime. Techniques include fuzz testing and taint analysis [Chen et al., 2017, Liu et al., 2008, Tang et al., 2010]. Hybrid approaches combine static and dynamic techniques [Liu et al., 2008].

Machine learning and deep learning techniques now dominate research in vulnerability prediction. These methods predict software vulnerabilities based on historical data and source code characteristics. Supervised learning, which involves training algorithms on labelled data, is a common approach in vulnerability prediction. Some common supervised learning algorithm examples include Decision Trees, Support Vector Machines (SVM), and Random Forests. Building a machine learning-based vulnerability prediction model involves collecting data, preparing the data, developing the model, and testing and evaluating it [Lin et al., 2020b]. Performance is typically evaluated using precision, recall, F1 score, and accuracy metrics. Challenges include the abundance of software bugs compared to vulnerabilities, making it difficult to find enough training data to effectively train models [Lin et al., 2020b, Ghaffarian and Shahriari, 2017]. Despite this, AI-driven techniques have outperformed rule-based methods, placing them at the forefront of vulnerability prediction research [Wang et al., 2021a]. Unsupervised learning, which identifies anomalies or outliers in unlabelled data, also plays a role, with examples including k-means clustering and anomaly detection algorithms [Ghaffarian and Shahriari, 2017]. Deep learning techniques, such as Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), and Long Short-Term Memory (LSTM) networks, have also gained traction in recent years [Lin et al., 2020a].

The following section delves into software vulnerability prediction, exploring the concept of granularity, vulnerability predictors (i.e., features or metrics), and source code representations used in vulnerability prediction research.

2.2 Software Vulnerability Prediction

Vulnerability prediction is a promising methodology in secure software engineering that identifies vulnerabilities before they can be exploited. Its primary goal is to help software testers allocate their limited resources efficiently by automatically identifying the most vulnerable components of a software system [Shen and Chen, 2020]. The main idea is to *preemptively* pinpoint software components that could be susceptible to security threats, thereby mitigating risks before attackers can exploit them. This research area has gained significant traction over the past decade [Hovsepyan et al., 2016]. Predicting which software components are likely to contain vulnerabilities promotes the early identification and mitigation of potential security issues during the development cycle [Shin et al., 2010].

Vulnerability prediction models are typically built using supervised machine learning techniques. These models utilise software attributes to differentiate between vulnerable and clean (or neutral) software components [Kalouptsoglou et al., 2023]. Recently, there has been a surge of interest in applying deep learning techniques to vulnerability prediction research [Lin et al., 2020a]. These models' primary goal is to identify software components more likely to contain vulnerabilities, enabling developers to concentrate on securing these critical areas [Shin et al., 2010].

A key aspect of vulnerability prediction is discerning the characteristics that differentiate vulnerable code components from non-vulnerable ones and other bugs. However, this task is challenging due to the inherent complexity of software systems. Coding-related vulnerabilities are often subtle and context-dependent, making them difficult to detect by casual observation. To overcome this challenge, vulnerability prediction approaches leverage machine learning techniques to identify vulnerable code patterns in software code. These patterns, often not apparent to human observers, make machine learning an ideal automated solution for detecting such subtle indicators. By employing machine learning, researchers aim to enhance the accuracy of vulnerability prediction and reduce the incidence of false positives.

2.2.1 Software Vulnerability Prediction Granularity

Artificial intelligence has been applied to vulnerability prediction to enhance software quality and reliability. The scarcity of testing and verification resources has driven security testers to develop methods to prioritise critical components, making vulnerability predictions essential for efficient resource allocation [Morrison et al., 2015].

Computational models for vulnerability prediction utilise historical software data and labelled vulnerable software artefacts. These models operate on source files to identify potentially vulnerable components. However, identifying vulnerabilities at the source code file level can be challenging, particularly with large files, as it complicates pinpointing the exact location of vulnerabilities. To address this issue, researchers have explored more granular prediction methods, such as method-level prediction, which allows for finer granularity in identifying vulnerabilities within software systems [Ghaffarian and Shahriari, 2017].

Besides source file-level and method-level granularity, other levels include binary-level, class-level, and change-level granularity [Kim et al., 2008]. Binary-level predictions are impractical due to the large number of files that require manual inspection after prediction. Source file-level granularity presents challenges with large files containing thousands of lines of code. Similarly, large classes with numerous methods pose difficulties at the class-level granularity. Change-level granularity, focusing on codebase changes, often lacks context, involving only code snippets. Consequently, method-level predictions are considered the most practical option [Morrison et al., 2015].

Giger et al. [2012] proposed a method-level bug prediction approach to reduce manual inspection steps. Method-level vulnerability prediction is analogous to method-level bug prediction, aiming to identify methods likely to contain vulnerabilities. Method-level vulnerability prediction in software security involves examining and assessing vulnerabilities at the level of individual methods or functions within a software codebase. This analysis focuses on scrutinising specific methods or functions to identify potential risks, using various techniques to predict the likelihood of vulnerabilities. This proactive approach aims to identify potential vulnerabilities at the method level before they manifest as security issues.

Method-level granularity in vulnerability prediction refers to the detailed analysis of potential security weaknesses at the level of individual methods within a codebase. Unlike coarser assessment levels, such as binary, file, or class levels, method-level granularity examines each method independently for vulnerabilities. Since methods are typically smaller than classes or source files, this approach enables more precise targeting of specific code sections where vulnerabilities may reside. It provides a more detailed understanding of potential risks associated with each method, facilitating localised identification of vulnerabilities and offering better insights into each method's security posture [Croft et al., 2022].

A detailed understanding of vulnerabilities at the method level would enable software professionals to develop more targeted security measures and remediation strategies. Such an approach could lead to more efficient resource allocation, ensuring that security efforts are concentrated where they are most needed at a much finer granularity, optimising the vulnerability mitigation process. Additionally, method-level vulnerability prediction could provide insights into the specific characteristics of vulnerable methods, aiding in the development of preventive measures and best practices to mitigate vulnerabilities in the future.

By incorporating method-level vulnerability prediction into the development workflow, software professionals can ensure the early identification of potential security issues, making the process more efficient than waiting until an entire class or source file is written. This strategy enables the more accurate identification of vulnerability sources, facilitating timely and effective remediation [Giger et al., 2012, Morrison et al., 2015, Sultana et al., 2023].

2.2.2 Software Vulnerability Prediction Features

The features used in vulnerability prediction models are crucial for their performance. These features are characteristics of software components that enable the model to distinguish between vulnerable and non-vulnerable components. As input to the machine learning model, features enable it to learn patterns indicative of vulnerabilities, making the careful selection of these features essential for accurate predictions.

Many studies have utilised traditional software metrics as features. These metrics, both product and process metrics, measure various aspects of software engineering products and processes. Product metrics quantify attributes such as code size and complexity, while process metrics measure factors like developers' productivity. When applied correctly, these metrics can define the success or failure of a software product or process, providing valuable information for making business decisions and improving software systems [Chhabra and Gupta, 2010].

Traditional software metrics are often calculated from components such as classes and methods. In the context of vulnerability prediction research, the goal is to leverage these metrics to identify components prone to vulnerabilities based on observations that vulnerable code components often exhibit specific characteristics. Therefore, careful selection of metrics can enhance the predictive performance of vulnerability prediction models [Singh et al., 2011].

There are two broad categories of software metrics:

- **Product Metrics:** These metrics quantify source code attributes, such as code size and complexity. Examples include Lines of Code (LOC) and McCabe's Cyclomatic Complexity.

- **Process Metrics:** These quantify software-related processes, such as the number of code changes (churn) or developer characteristics.

There has been a longstanding debate about which metric category performs better in vulnerability prediction [Rahman and Devanbu, 2013] and whether traditional software metrics alone are sufficient for accurate vulnerability prediction. Researchers have observed that vulnerable code tends to have specific characteristics: it is often large, complex, tightly coupled, and frequently churned [Giger et al., 2012, Morrison et al., 2015, Pascarella et al., 2018, Du et al., 2019]. These attributes can be measured using product software metrics. Traditional metrics, such as McCabe's Cyclomatic Complexity, Lines of Code, Code Churn, and Dependency metrics (Fan-In and Fan-Out), are commonly used as features in vulnerability prediction models. However, these metrics face several challenges that can limit their effectiveness in accurately predicting vulnerabilities.

The primary challenge is their limited ability to capture security-specific code characteristics. Traditional software metrics were initially designed to measure software quality and productivity, not security. Consequently, they may miss security-specific aspects, such as input validation, authentication, and authorisation. For instance, traditional metrics might deem a codebase with low complexity and churn secure, but in reality, it could contain vulnerabilities due to insufficient input validation. While traditional software metrics can be valuable as supplementary features, they may be insufficient for accurately predicting vulnerabilities. Therefore, vulnerability prediction models must incorporate features that capture security-relevant code attributes. Researchers argue that the inadequacy of traditional metrics arises from their reliance on syntactic code characteristics, which often lack semantic depth [Lin et al., 2020a]. Traditional metrics do not convey the underlying meaning of the code. This limitation is why many contemporary studies employ source code representation-based features, as they provide a more nuanced view of the code's syntax and semantics [Xiao et al., 2024]. Such features are more likely to identify subtle patterns distinguishing vulnerable code from non-vulnerable code. Consequently, studies using only traditional metrics often report poor results in vulnerability prediction [Ghaffarian and Shahriari, 2017].

In the next section, we delve into source code representations, examining the token-based and Abstract Syntax Tree (AST)-based representations employed in our research.

2.3 Source Code Representations

To accomplish different software engineering goals and improve software development and maintenance, practitioners have developed many methods, such as source code classification [Frantzeskou et al., 2008, Mou et al., 2016], code clone detection [Kamiya et al., 2002, Sajnani et al., 2016, White et al., 2016, Wei and Li, 2017], bug prediction [D'Ambros et al., 2012, Tantithamthavorn et al., 2016], and code summarisation [Haiduc et al., 2010, Jiang et al., 2017]. However, a primary challenge common to these methods is effectively representing source code to capture its syntactical and semantic information.

Source code representation abstracts low-level details, providing a higher-level view. It transforms text-based source code into a more abstract and structured form, capturing syntactical and semantic information [Zhang et al., 2019]. This new form finds applications in various software engineering tasks. The abstraction enables tasks that are difficult or impossible to perform directly on text-based source code, including machine learning and deep learning tasks. These algorithms typically require numerical inputs, and source code representation facilitates converting textual source code into a format they can process [Hancock and Khoshgoftaar, 2020]. Many machine learning tasks require transforming raw data into a processable format, known as feature engineering. Effective source code representation captures embedded syntactical and semantic information. Each representation method has its advantages and disadvantages. The choice of representation depends on the specific task and the desired level of abstraction. Notably, no single representation is suitable for all tasks, as different representations capture different aspects of source code [Samoaa et al., 2022].

The following subsections discuss the source code representation approaches used in our research, including token-based and AST-based representations.

2.3.1 Token-based Representations

Token-based representations are among the most common methods for representing source code. They involve tokenising source code into a sequence of tokens, where each token represents a specific syntactic element, such as keywords, identifiers, literals, and operators [Zhou et al., 2020]. Token-based representations are widely used in tasks such as code clone detection [Li et al., 2017], bug prediction [Choudhary and Singh, 2017], and code summarisation [Fowkes et al., 2017]. These representations are relatively simple and easy to generate, making them convenient for many software engineering tasks.

For example, consider the following Java code snippet:

Listing 2.1: A Simple Java Main Method

```java
        public static void main(String[] args) {
            System.out.println("Hello, World!");
        }
```

Tokenising this code snippet would result in the following sequence of tokens:

```
public static void main ( String [ ] args ) { System .
   ↪ out . println ( " Hello , World ! " ) ; }
```

Figure 2.1: Token Representation of the Method in Listing 2.1

2.3.1.1 N-Grams

N-grams are a common feature representation technique in Natural Language Processing (NLP). An N-gram is a contiguous sequence of N items from a given sample of text or speech. In NLP, these items are typically words, characters, or tokens. N-grams capture the local context of words in a text, providing information about the relationships between adjacent words.

Character-based N-grams help capture morphological information, while word-based N-grams help capture semantic information [Abdolahi and Zahedh, 2017, Dogra et al., 2022].

For instance, the character-based 2-gram representation of the sentence "Hello, World!" would be `["He", "el", "ll", "lo", "o,", ", ", " W", "Wo", "or", "rl", "ld", "d!"]`.

Its 3-gram representation would be `["Hel", "ell", "llo", "lo,", "o, ", ", W", "Wor", "orl", "rld", "ld!"]`.

2.3.1.2 Shingles (Word N-Grams)

Shingles, also known as word N-grams, are a common feature representation technique in text processing. A shingle is a sequence of N words from a given text sample. Shingles capture the local context of words in a text, providing information about the relationships between adjacent words. For instance, a 2-shingle (bigram) captures the relationship between two adjacent words, while a 3-shingle (trigram) captures the relationship between three adjacent words.

To illustrate, the sentence "The quick brown fox jumps over the lazy dog" can be represented as 2-grams: ["The quick", "quick brown", "brown fox", "fox jumps", "jumps over", "over the", "the lazy", "lazy dog"].

Its 3-gram representation would be: ["The quick brown", "quick brown fox", "brown fox jumps", "fox jumps over", "jumps over the", "over the lazy", "the lazy dog"].

In the context of source code, N-grams can capture the local context of tokens within a code snippet. For example, a 3-gram representation of the Java code snippet in Listing 2.1 would be:

```
public static void↔static void main↔void main (↔main (
    ↪ String↔( String [↔String [ ]↔[ ] args↔] args )↔
    ↪ args ) {↔) { System↔{ System .↔System . out↔.
    ↪ out .↔out . println↔. println (↔println ( "↔( "
    ↪ Hello↔" Hello ,↔Hello , World↔, World !↔World !
    ↪ "↔! " )↔" ) ;↔) ; }
```

Figure 2.2: 3-Gram Representation of the Method in Listing 2.1

As shown in Figure 2.2, the 3-gram representation captures the local context of tokens in the code snippet, providing information about the relationships between adjacent tokens and encoding the syntactical structure of the code.

2.3.2 Abstract Syntax Tree-Based Representations

An AST represents the abstract syntactic structure of source code [Baxter et al., 1998]. This tree-like structure captures the hierarchical relationships between syntactic elements, such as statements, expressions, and declarations. In ASTs, nodes represent syntactic constructs, such as method declarations, expressions, or statements, while edges represent relationships between nodes, including parent-child or sibling relationships [Fluri et al., 2007]. These nodes and edges form paths representing the code's syntactic structure.

ASTs are widely used in programming languages and software engineering tools. Compared to plain source code, ASTs are more abstract and do not include all details, such as punctuation and delimiters, thus not representing every aspect of the code's syntax [Veeramani et al., 2014]. However, ASTs describe the lexical information and syntactic structure, such as method names and control flow structures. ASTs play essential roles in tasks like source code search [Paul and Prakash, 1994], program repair [Weimer et al., 2009], and source code differencing [Falleri et al., 2014]. ASTs also play essential roles in various program analysis tasks, such as type checking, code generation, and refactoring, as well as in machine learning-based software engineering tasks like code completion and recommendation [Miller, 1995, Jiang et al., 2021, Sommerlad et al., 2008, Liu et al., 2022a].

To illustrate how ASTs represent source code, Figure 2.3 shows the AST of the method in Listing 2.1. The AST captures the hierarchical relationships between syntactic elements in the source code, providing a more abstract view of the code's syntactic structure. Each node in the tree represents a syntactic construct, such as a method declaration, an expression statement, or a string literal, in the code being represented.

The CompilationUnit node represents the entire code snippet, while the TypeDeclaration node represents the class containing the Main method. The MethodDeclaration node represents the Main method, with child nodes representing the method's modifiers, return type, name, parameters, and body. The Block node represents the method's body, containing an ExpressionStatement node representing the println statement. The MethodInvocation node represents the println method invocation, with child nodes representing the method's target, name, arguments, and string literal argument.

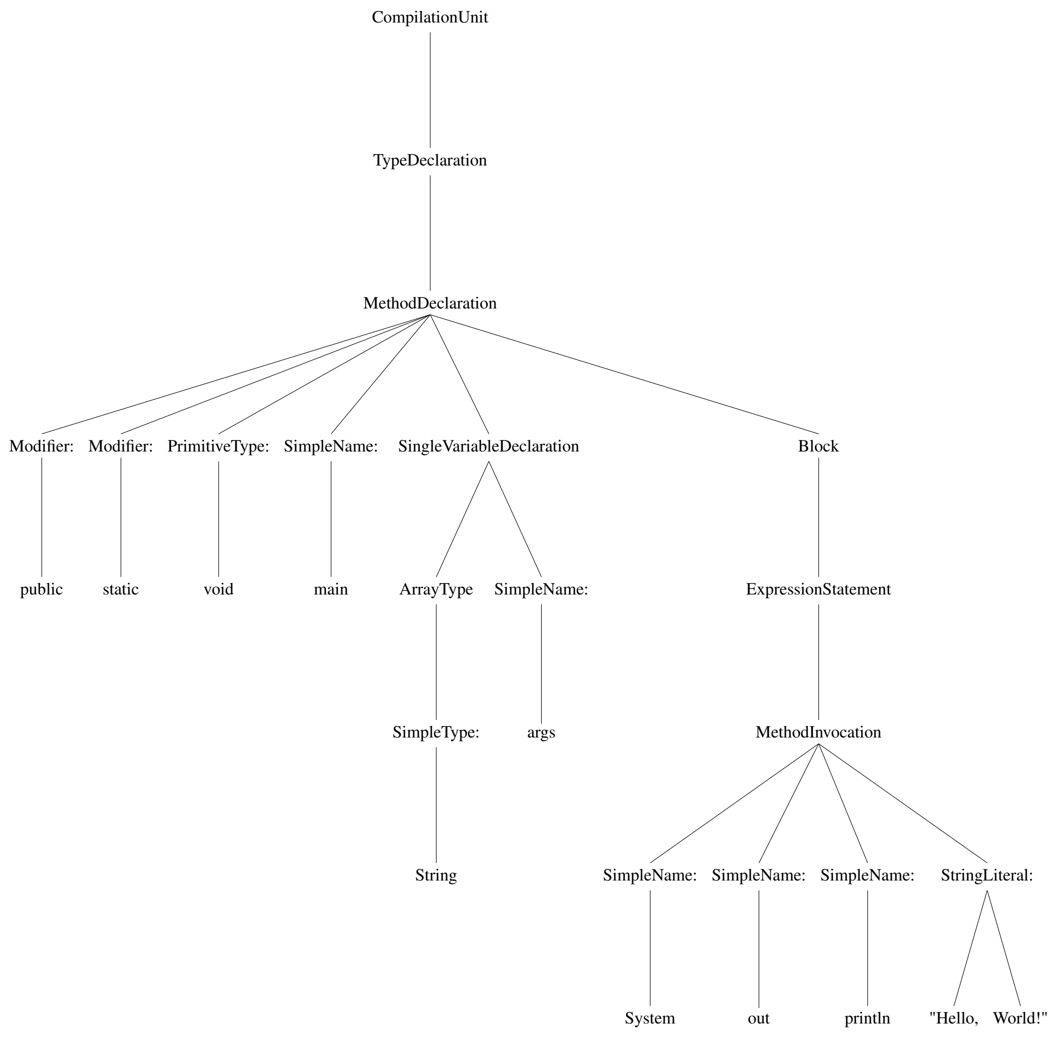


Figure 2.3: AST of the Method in Listing 2.1

ASTs form the foundational concept for AST-based representations, including Code2Vec, which utilises AST paths to represent source code. In this research, we use Code2Vec in our AST-based vulnerability prediction analyses.

2.3.2.1 Code2Vec Representation

Code2Vec is a source code embedding technique that extracts paths from the AST of the source code. It represents these paths as triples, comprising start nodes, end nodes, and internal path sequences. Code2Vec has been applied in various contexts since its introduction, notably in method name recommendation[1], as demonstrated by Alon et al. [2019].

[1]`https://code2vec.org/`

Listing 2.2: `getName` Method

```
        @Override
        public String getName() {
            return fileItem.getFieldName();
        }
```

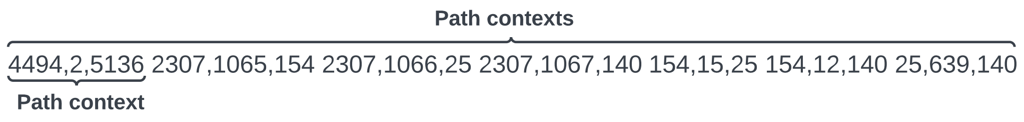


Figure 2.4: Code2Vec Representation for the Method in Listing 2.2

Listing 2.2 shows the source code of the `getName` method, and Figure 2.4 illustrates its Code2Vec (path-based) representation as extracted from the AST.

2.3.2.2 The Path Context Concept

Path-based representation models code snippets as collections of paths between AST nodes [Kovalenko et al., 2019]. The Code2Vec representation in Figure 2.4 includes seven whitespace-separated path contexts, a central innovation of Code2Vec.

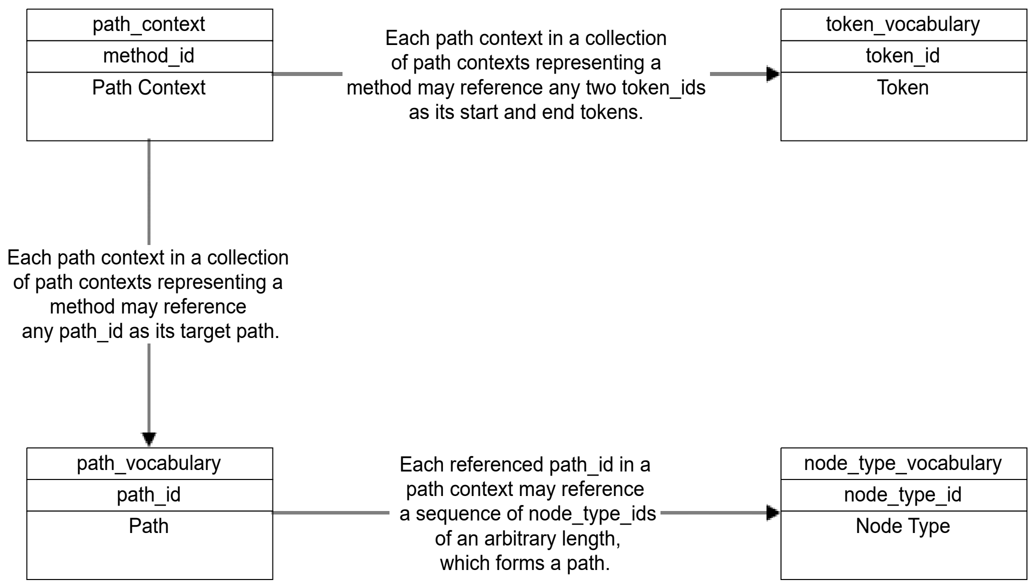

path_context
method_id
Path Context
Each path context in a collection of path contexts representing a method may reference any two token_ids as its start and end tokens.
token_vocabulary
token_id
Token
Each path context in a collection of path contexts representing a method may reference any path_id as its target path.
path_vocabulary
path_id
Path
Each referenced path_id in a path context may reference a sequence of node_type_ids of an arbitrary length, which forms a path.
node_type_vocabulary
node_type_id
Node Type

Figure 2.5: Path-Contexts: Representation of the Relationships among the Subcomponents

An ideal analogy to illustrate the path context concept is the Entity-Relationship Model (ERD). Figure 2.5 shows an ERD-like representation of the relationships among the path context subcomponents.

Suppose we have four entities (data files) representing the components of the path context concept:

1. *path_context*: This data file holds the Code2Vec representation (comprising path contexts) of methods, with a *method_id* column featuring unique numeric IDs that map to the path context of a method.

2. *token_vocabulary*: This data file holds tokens extracted from the source code, with a *token_id* column featuring unique numeric IDs that map to each token.

3. *node_type_vocabulary*: This data file holds the node types in the ASTs of the methods, with a *node_type_id* column featuring unique numeric IDs that map to each node type.

4. *path_vocabulary*: This data file holds sequences of node types forming paths in a method's AST, with a *path_id* column featuring unique numeric IDs that map to each path.

In this context, a path is a sequence of node types that indicates a traversal direction in an AST. A path context comprises a path enveloped by a start and end token, as shown in Figure 2.4.

Code2Vec captures the hierarchical structure and relationships within the source code by considering the code's context through path contexts. It learns connections between nodes within a path and relationships between nodes in different paths. This enables Code2Vec to effectively encode source code semantics, yielding robust representations for various software engineering tasks.

In Figure 2.4, each path context consists of a triplet: a path in the method's AST, surrounded by start and end tokens and separated by commas. The first number in each triplet is the start token, the second is the path, and the third is the end token.

For example, the annotated path context *4494, 2, 5136* means the start token references *token_id* 4494 in the *token_vocabulary*, the end token references *token_id* 5136, and the path references *path_id* 2 in the *path_vocabulary*, which in turn references a sequence of node types in the *node_type_vocabulary*.

token_id	**token**
4494	file|item
5136	get|field|name

(a) *token_vocabulary*

path_id	**path**
2	1 5 6 4

(b) *path_vocabulary*

node_type_id	**node_type**
1	SimpleName UP
4	SimpleName DOWN
5	METHOD_INVOCATION_RECEIVER UP
6	MethodInvocation TOP

(c) *node_type_vocabulary*

Figure 2.6: *token_vocabulary*, *path_vocabulary* and *node_type_vocabulary* Values for Path Context: *4494, 2, 5136*

Figure 2.6 contains three subtables detailing the interconnections among the components of the annotated path context in Figure 2.4, as derived from our data files. Other path contexts in Figure 2.4 and in other methods also exhibit similar interconnections, with the data files providing extensive information on tokens, paths, and node types, encompassing all source code elements within a software system or codebase.

2.3.3 Other Source Code Representations

Besides token-based and AST-based representations, other source code representations have been proposed for various software engineering tasks. For example, Control Flow Graphs (CFGs) represent a program's control flow in terms of basic blocks and their interconnections. CFGs are helpful in program analysis tasks involving reasoning about the program's execution behaviour, such as program slicing, optimisation, and program comprehension [Zhao et al., 2022, Anju et al., 2010].

Program Dependence Graphs (PDGs) are another technique for representing programs. They represent the data and control dependencies among program statements [Czech et al., 2017, Horwitz and Reps, 1992].

Many code representation techniques exist, and this section, by no means, covers all of them. As mentioned, each representation has advantages and disadvantages and is suitable for different tasks. Our research focuses on token-based and AST-based representations, which we use to construct features for vulnerability prediction models.

In the subsequent sections, we discuss the information retrieval techniques we employ to leverage these representations for vulnerability prediction.

2.4 Information Retrieval (IR)

The common idea behind the representations discussed in Section 2.3 is transforming source code into structured formats. These formats enable complex analyses and computations that would be challenging or impossible on raw data. They facilitate activities such as extracting meaningful features, applying machine learning, and measuring similarity and relevance for efficient searching, analysis, and comprehension. These applications are integral to our vulnerability prediction work. We combine source code representations with information retrieval techniques to implement these effectively. These techniques allow us to leverage these representations to retrieve relevant source code components, identify patterns, and measure similarity, which we use to construct features for our vulnerability prediction models.

2.4.1 Introduction to Information Retrieval

Information retrieval encompasses activities related to the organisation, processing, and accessing of information in various forms and formats. An information retrieval system enables a user to interact with an information system or service to find information, such as text, graphic images, sound recordings, or video, that meets their needs [Chowdhury, 2010].

Information retrieval is crucial for accessing information efficiently in the vast digital landscape. It facilitates decision-making, research, knowledge discovery, and numerous other applications across various fields and industries. It has applications in search engines, digital libraries, recommendation systems, e-commerce platforms, and other domains where accessing and retrieving relevant information is essential [Dong and Wang, 2008, Herrera-Viedma et al., 2008]. While web-based information retrieval is not the primary focus of our work, it remains the most visible application today because of search engines like Google, Bing, and Yahoo.

Many modern information retrieval systems handle multimedia information, including text, audio, images, and video. While features of conventional text retrieval systems apply to multimedia information retrieval, the unique nature of audio, image, and video information has necessitated the development of new tools and techniques. Modern information retrieval encompasses the storage, organisation, and access to text and multimedia resources [Chowdhury, 2010].

Despite its extensive applications, information retrieval remains a largely unexplored area in software security. We address this gap by applying information retrieval techniques to software security, with a specific focus on vulnerability prediction. Our information retrieval application primarily involves textual data, specifically data related to source code representation.

2.4.2 Information Retrieval System Features

Several features characterise information retrieval systems, essential for effectively organising, processing, and accessing information. The primary features include:

- **Indexing:** Information retrieval systems index documents to facilitate efficient searching. Indexing involves creating an index of terms extracted from the documents, allowing users to search for specific terms or phrases. This index provides a quick way to locate documents containing the desired information [Maron and Kuhns, 1960, Maron, 1977].

- **Querying:** Users interact with information retrieval systems by submitting queries. Queries are requests for information that users want to retrieve from the system. The system processes the queries and returns relevant documents based on the search criteria [Manning, 2008, Kobayashi and Takeda, 2000].

- **Ranking:** Information retrieval systems rank documents based on their relevance to the query. The ranking algorithm determines the order in which documents are presented to the user, with the most relevant documents appearing at the top of the search results [Pang et al., 2017, Jin et al., 2008].

- **Relevance Feedback:** Relevance feedback allows users to provide feedback on the relevance of search results. Users can indicate which documents are relevant or irrelevant, and the system uses this feedback to refine subsequent searches. This helps improve the accuracy of search results by incorporating user feedback [Lv and Zhai, 2009, Chang and Hsu, 1999].

- **Retrieval Models:** Information retrieval systems use retrieval models to determine the relevance of documents to a query. Retrieval models define how documents are scored and ranked based on their similarity to the query. Some standard retrieval models include vector space models, probabilistic models [Singhal et al., 2001], and language models [Song and Croft, 1999].

- **Evaluation:** Information retrieval systems are evaluated based on their ability to retrieve relevant information effectively. Evaluation metrics assess the system's effectiveness in retrieving relevant documents for users. Some standard evaluation metrics include precision, recall, F1 score, and mean average precision [Saracevic, 1995, Kobayashi and Takeda, 2000].

2.4.3 How We Use Information Retrieval in Our Research

Information retrieval plays a core role in our feature engineering process for vulnerability prediction. This thesis experiments with two types of source code representations, token-based and AST-based representations, to capture both the syntactical and semantic information of the source code we analyse. These representations enable us to extract meaningful features from source code for building models that predict vulnerabilities. By applying information retrieval techniques, we identify patterns in source code, quantitatively measure similarity between code components and retrieve relevant code snippets for analysis. These features help us predict vulnerabilities in software systems more accurately and efficiently.

2.4.3.1 Mapping Information Retrieval Features to Our Setup

Subsection 2.4.2 outlined the primary features of information retrieval systems, including indexing, querying, ranking, relevance feedback, retrieval models, and evaluation. Indexing, querying, and ranking are the most relevant to our research.

We use indexing to create an index of terms comprising elements from our code representations, as extracted from source code, enabling efficient searching. Querying is employed to process queries and retrieve relevant source code components. Ranking determines the relevance of these components to a query, ensuring that the most relevant components are prioritised. Our work does not involve relevance feedback and retrieval models. Additionally, while we evaluate the performance of our vulnerability prediction models, we do not assess our information retrieval setup, as it serves only as a means to an end rather than the primary focus of our research. Therefore, our work does not involve evaluation in the context of information retrieval systems.

Our datasets comprise a target software system, where we aim to predict vulnerabilities, and a vulnerability dataset, comprising known vulnerable code samples. These two datasets comprise source code representations of methods from the datasets. These representations are token-based and AST-based data derived from the source code of the methods. We index the source code representations of the methods in the vulnerability dataset to create an index of terms, allowing efficient searching of source code representation components. We then use the source code representations of the methods in the target software system as queries to retrieve relevant components from the vulnerability dataset. Finally, we rank the retrieved source code representation components based on their relevance to the query, ensuring that the most relevant components are presented first. The returned results and the top-ranked components are then used to construct features for our vulnerability prediction models.

The primary goal of our information retrieval setup is pattern matching and similarity measurement between source code components across the target software system and the vulnerability dataset. We aim to identify patterns in our target software system's source code that are consistent with known vulnerable code samples in the vulnerability dataset. These patterns indicate potential vulnerabilities consistent with the known vulnerabilities in the vulnerability dataset, enabling us to predict vulnerabilities more accurately and efficiently in the target software system.

2.4.3.2 Feature Engineering

Feature engineering is pivotal in machine learning and data science. It involves transforming raw data into meaningful features for predictive modelling. We employed information retrieval techniques to extract features from source code representations for vulnerability prediction, capturing syntactical and semantic information to enhance prediction accuracy.

Our work focuses on two primary feature types: token-based and AST-based, each of which is subdivided into two categories. The first, *hit-independent metrics*, consists of features derived directly from the intrinsic attributes of source code representations. The second, *hit-dependent metrics*, relies on information retrieval to measure the similarity between a method in the target software system and the most similar methods in a vulnerability dataset.

In our context, a *hit* occurs when a method in the target software system matches at least one method in the vulnerability dataset. Following a hit, features are used to quantify the similarity between the target software system method and the most similar method in the vulnerability dataset, as well as the general distribution of similarities between the target software system method and all methods in the vulnerability dataset.

Subsequent chapters will elaborate on our information retrieval-driven approach, detailing the extraction of source code representations, feature construction, and their application in machine learning models for vulnerability prediction. The immediate chapter, however, will review relevant literature, focusing on vulnerability prediction, general bug prediction and other related topics.

Chapter 3

Literature Review

This chapter presents a literature review on software vulnerability and bug prediction methodologies. It discusses critical studies that influenced our work and emphasises the role of machine learning and deep learning in improving predictive capabilities. The chapter concludes by discussing our observations in the field, identifying gaps in the literature, and laying the groundwork for the subsequent chapters.

3.1 Introduction

Previous chapters have highlighted the importance of understanding the adverse impact of software vulnerabilities in modern development. As reliance on digital infrastructure grows, so do the risks associated with software vulnerabilities. This chapter presents a comprehensive literature review, which is essential for contextualising current research, identifying critical methodologies, and highlighting significant findings in software vulnerability and bug prediction.

The proliferation of software vulnerabilities presents an ongoing challenge in maintaining secure systems. The increasing complexity of software and rapid development have made identifying and mitigating vulnerabilities more challenging. Traditional detection methods, such as static and dynamic analysis, provide valuable insights but face limitations in terms of accuracy and scalability, particularly in modern systems. These challenges have driven interest in advanced methodologies, notably those using machine learning, to enhance predictive capabilities.

Predicting software vulnerabilities is a proactive approach to identifying potential threats before they can be exploited. It involves analysing historical data and source code characteristics to detect patterns indicative of vulnerabilities. The literature reveals various techniques, ranging from simple to sophisticated, that collectively enhance our understanding of how vulnerabilities manifest and propagate, ultimately leading to the development of innovative mitigation strategies.

The advent of machine learning and deep learning has transformed vulnerability prediction. Supervised learning techniques show promise in identifying vulnerability-prone components by learning from labelled datasets. These techniques use various features from the source code, including syntactic and semantic information, to train predictive models. Deep learning further enhances these capabilities by capturing complex patterns within the code.

Despite advancements, challenges remain. High false positive rates in many predictive models lead to inefficiencies in vulnerability prediction. Additionally, the scarcity of comprehensive labelled datasets hinders the development and evaluation of robust models.

This literature review highlights the critical need for innovative approaches in secure software development. It sets the stage for subsequent chapters, which delve deeper into specific methodologies and experimental evaluations undertaken in this research. The findings highlight the dynamic nature of the field and ongoing efforts to enhance software security through predictive analytics.

This chapter organises the reviewed literature into two main themes: software vulnerability prediction and software bug prediction. The first theme focuses on studies of software vulnerabilities and prediction methodologies, with an emphasis on machine learning and deep learning techniques. The second theme explores general (non-security-relevant) software bug prediction methodologies, providing a broader context for understanding software quality and reliability.

We highlight studies that significantly influenced our work within each theme and also review relevant systematic reviews and comparative studies to provide a comprehensive overview of the field.

The chapter concludes with a discussion of our observations and emphasises the importance of identifying gaps in the literature. These gaps present opportunities for future research and highlight the novelty and significance of our work in predicting software vulnerabilities.

3.2 Software Vulnerability Prediction

This section reviews software vulnerabilities and the methodologies developed to predict them. It discusses critical studies that have influenced our work and highlights the role of machine learning and deep learning in enhancing predictive capabilities. It also considers systematic reviews and comparative studies to provide a comprehensive overview of the field, placing our research within the broader context of software vulnerability prediction.

3.2.1 Vulnerability Prediction Studies Influencing Work

This subsection reviews critical vulnerability prediction studies that significantly influenced our work and laid the foundation for our research. Their innovative methodologies, insightful findings, and contributions to the field of software vulnerability prediction influenced our work.

Shin et al. [2010] sought to enhance tools for security testers by exploring the predictive potential of software metrics data, including source code characteristics and historical data. They examined metrics related to complexity, code churn, and developer activity in empirical studies of Mozilla Firefox and Red Hat Enterprise Linux kernel. Their results indicated that 24 out of 28 metrics effectively distinguished between vulnerable and non-vulnerable files in both projects. Machine learning models that incorporated all three categories of metrics predicted over 80% of the pre-identified vulnerable files, with fewer than 25% false positives in both cases. The study concluded that these models could significantly reduce inspection efforts by up to 71% for Mozilla Firefox and 28% for Red Hat Enterprise Linux.

Hovsepyan et al. [2012] critiqued traditional software vulnerability prediction methods that rely on 'cooked' features, such as code complexity and churn. They introduced an alternative approach using raw source code analysis as text to develop features. Their model, tested across 18 versions of a large mobile application, achieved an average accuracy of 0.87, a precision of 0.85, and a recall of 0.88. The study concluded that this method is a viable complement to software metrics-based methods and highlighted a future research direction that combines textual source code analysis with software metrics as features.

Meneely et al. [2013] pointed out that software security is crucial in modern development, as vulnerabilities, often stemming from design and coding flaws, can linger undetected, posing substantial risks. The researchers aimed to enhance security by examining the size, churn, and community dissemination of Vulnerability-Contributing Changes (VCCs). They employed a hybrid approach to trace 124 commits that resulted in 68 known vulnerabilities in the Apache HTTPD server, some of which dated back nearly two decades. They traced these VCCs using the 'git bisect' command. The analysis revealed that VCCs are generally larger than non-VCCs and that vulnerabilities are more likely to be introduced by new developers working on specific sections of the code. The findings suggest that understanding these patterns can help developers reduce the risk of introducing vulnerabilities.

Shin and Williams [2013] explored whether fault prediction models could effectively predict vulnerabilities or if dedicated Vulnerability Prediction Models (VPMs) were necessary. Their study, which used traditional software metrics such as complexity, code churn, and fault history, focused on the Mozilla Firefox web browser. They found that while 21% of the source code files contained faults, only 3% contained vulnerabilities. Their analysis showed that fault and vulnerability prediction models performed comparably at various classification thresholds. For instance, at a threshold of 0.6, the fault prediction model achieved a recall of 83% and a precision of 11%; similarly, the VPM reached a recall of 83% and a precision of 12% at a threshold of 0.5. They concluded that software metrics-based fault prediction models could also serve as effective VPMs. However, both model types require further refinement to decrease false positives and enhance recall.

Morrison et al. [2015] explored the challenges of implementing VPMs in large-scale systems, such as the Windows Operating System, highlighting their role in helping software engineers prioritise verification resources. The study evaluated the accuracy and actionability, which is defined as the inspection effort required to assess the results of VPMs constructed using standard recommendations. The researchers replicated a VPM for two Windows releases, adjusting model granularity and statistical learners, and assessed the models' precision, recall, and required inspection effort for security reviews. The findings indicated that while binary-level predictions offered high precision (0.75), they required an impractically high inspection effort due to low recall (approximately 0.20). On the other hand, source file-level models showed lower precision (less than 0.5) and recall (less than 0.2). These outcomes suggest that VPMs need further refinement to become actionable, possibly by integrating security-specific metrics. The study concluded that although VPMs are promising, significant enhancements are necessary to improve their precision and recall. It recommended further research and the inclusion of security-specific metrics.

Al Debeyan et al. [2022] highlighted the criticality of the Log4jshell vulnerability[1] to highlight the importance of effective vulnerability detection in software systems. They critiqued traditional vulnerability prediction models for their binary classification approach, which only identifies if a code component is vulnerable. They argued that these models should also inform developers about the nature of the vulnerabilities they detect. To enhance this, they developed a multiclass classification model that categorises vulnerabilities by type. They analysed vulnerable and non-vulnerable methods, extracting vulnerability types and decomposing the code's Abstract Syntax Trees (ASTs) into n-grams. This data was used to train classifiers, and their random forest model achieved an F measure of 75% and a Matthews Correlation Coefficient (MCC) of 74%. The study concluded that using n-grams from ASTs to train classifiers offers a more insightful approach to vulnerability prediction, as it provides detailed information on vulnerability types.

3.2.2 Machine Learning-Based Vulnerability Prediction

Machine learning has revolutionised vulnerability prediction by utilising historical data and source code features to develop predictive models. This subsection re-

[1]https://www.ncsc.gov.uk/information/log4j-vulnerability-what-everyone-needs-to-know

views studies that applied machine learning techniques, including supervised, semi-supervised, and unsupervised learning, to vulnerability prediction.

3.2.2.1 Supervised Learning-Based Vulnerability Prediction

Although in-code vulnerabilities can be complex, context-dependent, and challenging to detect, they inherently conform to finite patterns because they are based on specific code structures. Therefore, supervised learning techniques are well-suited to predict vulnerabilities by learning from labelled data. This subsubsection reviews studies that have applied supervised learning to predict software vulnerabilities.

Younis and Malaiya [2014] emphasised the growing risk posed by the increasing number of software vulnerabilities and the shrinking timeframe between their public disclosure and exploitation. They stated that traditional methods for assessing vulnerability exploitability often rely on subjective judgment and are not scalable, underscoring the need for automated, scalable, and objective methods. To address this, the study proposed a new metric based on software structure properties such as attack entry points, vulnerability locations, dangerous system calls, and reachability. Their research aimed to introduce and evaluate this new metric, which they designed to reduce subjectivity and enhance scalability in assessing exploitation risks. Additionally, they developed a Support Vector Machines (SVM) model using this metric to automatically predict the risk of vulnerability exploitation. The model's effectiveness was tested using data from the National Vulnerability Database and various exploit databases. Preliminary results suggest that factors such as reachability and the presence of dangerous system calls strongly predict exploitability. The model showed promising results in distinguishing between exploitable and non-exploitable vulnerabilities, outperforming traditional methods. The study highlighted the potential of integrating software structure properties and machine learning to automate and improve the accuracy of vulnerability exploitation assessments. The study concluded that their approach could significantly help prioritise vulnerabilities and allocate resources more effectively, offering a valuable direction for future research in vulnerability assessment.

Perl et al. [2015] stated that security experts recognise the exponential increase in vulnerabilities despite intensified efforts to mitigate them. They pointed out that pre-deployment security audits struggle with the impracticality posed by the vast volume of code. In contrast, tools like Flawfinder, a Python-based tool that helps identify vulnerabilities in C/C++ source code, although helpful in identifying vulnerabilities, suffer from high false positive rates. To improve this, the researchers introduced VCCFinder, a tool designed to identify vulnerable code in repositories with significantly lower false positive rates. They linked Common Vulnerabilities and Exposures (CVE) entries to corresponding GitHub commits, creating a database comprising 66 C and C++ projects with 170,860 commits. They used a crawler to identify vulnerable commits by searching for CVE IDs in commit messages and validated their accuracy through manual analysis of a 10% random sample. A Support Vector Machines model trained using this database classified vulnerable commits. The study reported that VCCFinder dramatically reduced the false positive rate by up to 99% compared to Flawfinder, producing only 36 false positives against Flawfinder's 5,460. The research highlighted the potential for further research to generalise their findings and develop recommendations for reducing vulnerabilities in deployed software systems.

Li et al. [2016] stated that the rapid increase in software vulnerabilities poses significant challenges in cybersecurity, as these vulnerabilities are often exploited in various types of attacks. The researchers stated that patching vulnerabilities is crucial, but reusing vulnerable code across different software versions and libraries complicates vulnerability tracking and patching efforts. Existing tools frequently fail to identify all instances of a vulnerability, highlighting the need for more effective automated detection systems. They introduced VulPecker, an automated system that detects software vulnerabilities using code similarity analysis. This system aims to overcome the limitations of current detection methods by identifying vulnerable code fragments across diverse software products and versions. VulPecker utilises a Vulnerability Patch Database (VPD) and a Vulnerability Code Instance Database (VCID), which store information about known vulnerabilities and their code instances. It characterises patches using defined features and applies various code similarity algorithms to detect vulnerabilities. Classifiers are built to determine the most effective algorithm for each vulnerability type. The researchers evaluated VulPecker, demonstrating its ability to detect 40 vulnerabilities not recorded in the National Vulnerability Database (NVD), including 18 previously unknown vulnerabilities and 22 vulnerabilities that were silently patched in subsequent releases. The system significantly improved detection accuracy, achieving high precision and recall. The researchers attributed its success to selecting appropriate code similarity algorithms for different types of vulnerabilities. They then suggested that future research could extend its application to additional programming languages and explore scalability for larger software systems. This study highlighted the potential of automated systems to enhance cybersecurity by enabling more thorough and accurate detection of vulnerabilities and patching.

Sultana [2017a] emphasised the crucial role of software security in maintaining overall software quality, noting the rising number of vulnerabilities and the sophistication of attacks. They critiqued traditional software metrics for their high false negative rates and lack of specific guidance on secure coding. To overcome these limitations, the study aimed to develop a more effective software vulnerability prediction model by introducing the concept of traceable patterns, which can be automatically recognised and extracted from source code. The research explored whether traceable patterns could more accurately predict vulnerable code than traditional metrics by analysing class-level and method-level patterns. Data from projects like Apache Tomcat and various Java web applications were used. Machine learning and statistical methods were employed to predict vulnerabilities using features derived from traceable patterns and traditional metrics. The findings revealed that traceable patterns yielded a lower false negative rate and higher recall compared to traditional metrics. Patterns like CompoundBox and Immutable indicated secure code, whereas Outline and AugmentedType were associated with vulnerability. Their Support Vector Machines models demonstrated enhanced performance in predicting vulnerabilities when trained with traceable patterns, presenting a substantial advancement in predicting software vulnerabilities. The study advocated for integrating traceable patterns in vulnerability assessments to give developers better tools for prioritising code reviews and testing efforts. They recommended that future research focus on refining these patterns and testing their applicability in different programming languages and development environments.

Sultana and Williams [2017] focused on enhancing the early detection of vulnerable code to improve the cost-effectiveness and efficiency of software testing. They explored the utility of class-level traceable patterns, known as micropatterns, which can be automatically mined from source code. The study hypothesised that micropatterns could outperform traditional software metrics in vulnerability prediction. To verify this, they compared the effectiveness of micropatterns against conventional class-level metrics in predicting vulnerabilities. Machine learning techniques were applied using data from Java-based systems, such as Apache Tomcat 7 and Apache Camel, with micropatterns and class metrics as features. The results demonstrated that micropatterns achieved higher recall in identifying vulnerable classes than traditional metrics. Specifically, while class metrics showed precision values ranging from 0.693 to 0.995 and recall values ranging from 0.250 to 0.786, micropatterns, analysed using a Decision Tree algorithm, achieved precision values between 0.604 and 0.705 and recall values between 0.875 and 0.923. The study concluded that micropattern-based models are more effective at predicting vulnerable Java classes than models based on traditional class-level metrics, suggesting a significant potential for improving vulnerability detection methodologies.

Yang et al. [2017] stated that predicting software vulnerabilities before code audits is ideal. However, many models lack granularity because they typically operate at file or component levels, which can be costly and impractical. To address this, they introduced VulDigger, a code review tool for identifying potentially vulnerable commits in software systems' commit histories at the change level. The study developed a dataset from Mozilla Firefox and used a classification tool that leverages metrics from bug and vulnerability detection. They adopted a unique approach similar to Perl et al. [2015]'s work (also reviewed) for semi-automatically mapping vulnerabilities to contributing changes. Their approach achieved a high precision of 0.92 but a low recall of 0.14. The study concluded that this method supports continuous security inspection by providing instant feedback on code changes, aiding in more targeted and efficient vulnerability management.

Sultana et al. [2018a] aimed to enhance early vulnerability detection in software development, noting the limitations of traditional metrics in accurately locating vulnerabilities. They investigated the efficacy of method-level traceable patterns (nano-patterns) compared to traditional software metrics in vulnerability detection. Their research involved experiments using Apache Tomcat 6 and 7, Apache CXF, and two standalone Java web applications. They employed three machine learning techniques to evaluate nano-patterns and method-level metrics. The results indicated that nano-patterns provided lower false-negative rates and higher recall values than traditional metrics, making them more effective for identifying vulnerable code. Specifically, nano-patterns achieved precision between 0.676 and 0.812 and recall between 0.689 and 0.897, while traditional metrics showed higher precision between 0.775 and 0.871 but lower recall between 0.500 and 0.776. The study concluded by recommending that developers integrate nano-patterns as features in vulnerability prediction models to improve detection accuracy.

Jimenez et al. [2019] critiqued the common assumption in vulnerability prediction research that models are trained with sufficient and accurate labelling. They noted that this idea does not reflect real-world scenarios, where data is often partial and mislabelled, pointing out that this discrepancy can lead to promising empirical results that diminish under realistic conditions. To explore this, they analysed 1,898 real-world vulnerabilities across 74 releases of Linux Kernel, OpenSSL, and Wireshark, evaluating the effectiveness of three vulnerability prediction approaches, both with and without the unrealistic labelling assumption. Their findings showed that unrealistic labelling could significantly skew results, with Matthews Correlation Coefficient values of 0.77, 0.65, and 0.43 for Linux Kernel, OpenSSL, and Wireshark, respectively, under unrealistic conditions. When using more realistic labelling, these values fell to 0.08, 0.22, and 0.10. The study concluded that vulnerability prediction research needs to improve experimental and empirical methods to ensure the robustness and practical applicability of the findings.

Chong et al. [2019a] discussed the escalating threat posed by the quick transition from discovery to exploitation of software vulnerabilities. They highlighted the inefficiencies of traditional vulnerability detection methods and explored the use of software metrics as predictors for vulnerable code components in open-source Java and Python projects. The aim was to assess the predictive capability of these metrics and compare their effectiveness across these programming languages. The study focused on function-level metrics like Cyclomatic Complexity, Lines of Code (LOC), and Nesting Levels. The research employed machine learning classifiers, including Support Vector Machines and Logistic Regression (LR), to analyse vulnerabilities in three Java projects (Apache Tomcat 6, Tomcat 7, and Apache CXF) and two Python projects (Django and Keystone). The models were evaluated based on precision, recall, and false positive rates. The findings indicated that software metrics were more effective in predicting vulnerabilities in Java projects, with SVM models achieving a recall of 70-73% and LR models about 67-71%. In contrast, Python projects saw lower recall rates, with SVM at 42-50% and LR at 40-46%. This suggests that while software metrics are reliable predictors for Java, their effectiveness in Python requires enhancement. The study concluded that software metrics hold significant potential as predictors of vulnerable code components, especially in Java. However, the varied performance across programming languages indicates a need for further research to optimise the use of software metrics in different coding environments. The researchers recommended investigating additional metrics and advanced machine learning techniques to improve vulnerability prediction across all domains.

Sultana et al. [2021] stated that maintaining software systems to mitigate vulnerabilities is crucial, especially as some vulnerabilities can remain dormant for years and persist across software releases. In this context, the researchers focused on how effectively software metrics can predict vulnerabilities at different granularities. They conducted a comparative study using class-level and method-level metrics across four Java projects to train supervised learning models. These metrics included traditional process indicators such as code size, complexity, and nesting. The study found that class-level granularity achieved a recall of over 70% and a precision of over 75%, while method-level metrics yielded recall and precision values of over 65% and 80%, respectively. These results suggest that targeted testing based on these metrics can effectively mitigate the risks associated with dormant vulnerabilities. The researchers highlighted the potential of using such metrics to guide development teams in focusing their testing efforts on the most susceptible areas of the code.

Zhou et al. [2021] highlighted the importance of secure programming practices in mitigating vulnerabilities during software development. They stated that while many studies have developed vulnerability prediction models using software metrics, the impact of vulnerability fixes on these metrics has been less explored. The researchers examined how specific metrics change following vulnerability fixes, using static analysis to compare metrics before and after fixes for 250 vulnerable files from Apache Tomcat and Apache CXF. The analysis focused on metrics such as class and method counts, variable instances, maximum nesting, lines of code, and complexity. They observed a minimum increase of 2% in metrics including *CountDeclClass*, *CountDeclClassMethod*, *CountDeclClassVariable*, *CountDeclInstanceVariable*, *CountDeclMethodDefault*, *CountLineCode*, *MaxCyclomaticStrict*, and *MaxNesting* post-fix. The findings suggest that understanding the effect of fixes on these metrics could guide the development of more effective vulnerability prediction models.

Recognising the need for secure software development, Medeiros et al. [2021] conducted two experiments to enhance software security early in the development lifecycle. The first experiment focused on developing vulnerability prediction models using software metrics calculated from the Linux Kernel and Mozilla Firefox, including McCabe's Cyclomatic Complexity, Lines of Code, Coupling Between Objects, and Lack of Cohesion. They trained five different machine learning models using these metrics for vulnerability detection. The second experiment introduced a consensus-based decision-making approach to categorise code components by their perceived vulnerability into four categories: 'Highly Critical', 'Critical', 'Low Critical', and 'Non-Critical '. This method combined the classification results from the first experiment to assess the potential vulnerability of code components. The study found that the consensus-based method was more effective across various development scenarios than standalone vulnerability prediction models. It concluded that while software metrics alone may not be sufficient to identify vulnerable code due to high false positives, they are valuable in scenarios where precise vulnerability detection is crucial.

Ganesh et al. [2021] stated that organisations often deploy open-source systems to manage sensitive data, necessitating rigorous security checks early in development to prevent cyberattacks. Recognising the need for tools that enable developers to identify vulnerabilities during coding, they assessed the utility of machine learning algorithms in detecting potentially vulnerable software components via source code analysis. They sourced security vulnerability data and source code for Apache Tomcat versions 4 to 10 to compute 43 object-oriented metrics, such as coupling, cohesion, and complexity, which formed the basis of their model features. The research utilised Naïve Bayes, Decision Tree, K-Nearest Neighbors (KNN), and Logistic Regression. It found that the KNN algorithm achieved the highest accuracy at 80%. However, the researchers cautioned that these promising results are specific to Apache Tomcat and may not apply universally across different software systems.

Pereira et al. [2021] highlighted the unreliability of many vulnerability detection methods used by software developers, noting exceptionally high false-positive rates. They proposed a combined approach using static analysis tool (SAT) alerts and software metrics to enhance vulnerability detection in the Firefox Mozilla project, which is written in C and C++. They constructed datasets incorporating SAT alerts, software metrics, and their combination. After that, they conducted several classification experiments, including binary classification, binary per category, and multiclass classification using Decision Trees, Random Forests, Extreme Gradient Boosting, and Bagging algorithms. The results demonstrated that the Bagging algorithm, utilising software metrics data, achieved the highest precision of 0.94, while the combination of SAT alerts and software metrics reached the highest recall of 0.90. The best F1 measure was 0.36, using software metrics with Bagging. The findings indicated that software metrics generally outperform SAT alerts in machine learning models and highlighted the difficulty of simultaneously achieving high precision and recall. The research revealed that vulnerable and non-vulnerable files often have similar attributes, complicating the differentiation process.

Hocking et al. [2022] highlighted the crucial role of computers in modern society but emphasised the growing threat of cyberattacks exploiting vulnerabilities, thereby highlighting the need for practical vulnerability prediction tools. Their research focused on methods that treat source code as text files, particularly interpretability analysis, to identify the most critical features in predicting vulnerabilities. The study involved developing features from over 2.4 million C and Java components to predict vulnerability proneness. They employed an L1-regularised logistic regression model, known for its interpretability due to the Lasso/L1 regularisation technique, and used a Gradient Boosting algorithm as a non-linear baseline. They also explored combining neural network embedding features with L1-regularised models. A 10-fold cross-validation revealed that linear models with interpretable features outperformed those relying solely on neural network embeddings. The study concluded that combining interpretable and neural network embedding features is essential for optimal prediction performance. However, they acknowledged the study's limitation in considering only one interpretable feature, code complexity, and suggested expanding the feature set to include code churn and developer activity in future research.

Le and Babar [2022] noted that while many researchers have explored vulnerability detection in program functions and fine-grained code statements, few have focused on using the output of these methods to assess vulnerabilities and gain deeper insights into their nature, which is critical for vulnerability prioritisation. To address this gap, they investigated automating function-level vulnerability assessment. Their dataset included 1,782 functions with 429 vulnerabilities from 200 real-world software systems. They gathered vulnerable and non-vulnerable code statements from vulnerability-fixing commits sourced from the National Vulnerability Database, GitHub Advisory Database, and VulasDB. The researchers extracted the context of these vulnerable statements to generate features, which were then input into six classifiers to develop models for predicting Common Vulnerability Scoring System (CVSS) metrics. The study achieved a maximum performance of 0.64 Matthews Correlation Coefficient and 0.75 F1 score. The authors concluded that further research is needed in function-level vulnerability assessment, with a primary focus on techniques that effectively capture the relationships between vulnerable and non-vulnerable code statements.

Napier et al. [2023] highlighted the increasing difficulty of resolving software vulnerabilities due to their growing complexity and severity. They added that while traditional vulnerability detection methods are valuable, machine learning-based approaches are gaining prominence. To evaluate the effectiveness of these methods, the researchers conducted a study focusing on text-based machine learning models. The study utilised a dataset of 344 open-source projects comprising 2,182 vulnerabilities and 38 vulnerability types. To address the class imbalance, they extracted functions from the source code and applied a pairing technique between fixed and vulnerable versions of the same code samples. They tested seven machine learning models and various Natural Language Processing and data processing techniques. The experiments detected vulnerabilities within and across the top 10 projects and the top 10 CWE vulnerability types based on the number of extracted function pairs. After statistical analysis, the average precision ranged from 52.07% to 63.36% within and across the top 10 projects and from 55.19% to 61.61% within and across the top 10 CWE types. The study concluded that text-based machine learning vulnerability detectors are ineffective for detecting vulnerabilities across projects and CWE types.

3.2.2.2 Semi-Supervised Learning-Based Vulnerability Prediction

Semi-supervised learning is less explored in vulnerability prediction compared to supervised learning. This subsubsection reviews a study that revisits a prior supervised learning-based approach using a semi-supervised technique.

VCCFinder, introduced by Perl et al. [2015] (also reviewed), is a machine learning-based methodology for detecting vulnerabilities through code change analysis. Timothé et al. [2021] attempted to replicate VCCFinder's supervised learning approach but faced challenges due to the unavailability of the original resources. Consequently, they developed an alternative method using a semi-supervised learning technique and a different set of features; their study also explored the difficulties in identifying vulnerability-contributing commits, which often lack explicit tags or messages. The alternative approach did not yield the same results as VCCFinder, indicating a gap in replicability and effectiveness. Despite this, the authors regarded their findings as a constructive baseline for future research in vulnerability prediction, highlighting the field's ongoing challenges.

3.2.2.3 Unsupervised Learning-Based Vulnerability Prediction

Like semi-supervised learning, unsupervised learning is less commonly used in vulnerability prediction. This subsubsection reviews studies that explore unsupervised approaches, particularly anomaly detection-based vulnerability prediction.

To enhance security testing, Yamaguchi et al. [2013] developed Chucky, an anomaly detection tool designed to identify missing checks in source code. Recognising that many vulnerabilities stem from insufficient input validation, they used Chucky to statically analyse the code for omitted conditions related to security-relevant objects. The tool was tested on projects including Firefox, Linux, LibPNG, LibTIFF, and Pidgin, discovering 12 previously unknown vulnerabilities in the latter two. The researchers also suggested that Chucky could be integrated with techniques like fuzzing or symbolic execution to further analyse and rank the severity of these vulnerabilities. Using anomaly detection to identify vulnerabilities, this approach marked a notable advancement in unsupervised vulnerability identification approaches.

In response to increasing cyberattacks targeting the Google Android platform, Malik et al. [2019] conducted a study to detect anomalies in system calls within Android applications, aiming to distinguish between benign and malicious behaviours. The study hypothesised that the type, frequency, and sequence of system calls linked to vulnerabilities exhibit distinct patterns. Using machine learning techniques, they employed several metrics and parameters to monitor system processes and differentiate between normal and harmful activities. Their K-Nearest Neighbours algorithm achieved a precision of 0.852, a recall of 0.839, and an F1 score of 0.846. The Long Short-Term Memory (LSTM) algorithm demonstrated a precision of 0.786, a recall of 0.946, and an F1 score of 0.856. Additionally, an enhanced LSTM Genetic Algorithm achieved a precision of 0.752, a recall of 0.988, and an F1 score of 0.854. The study concluded that these machine learning approaches could effectively predict bugs and vulnerabilities, with an F score of around 85%. This study also prioritised anomaly detection techniques, similar to Yamaguchi et al. [2013] (also reviewed).

3.2.3 Deep Learning-Based Vulnerability Prediction

This subsection reviews studies that have applied deep learning techniques, including Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), and LSTM networks, to predict software vulnerabilities.

Lin et al. [2018] stated that security testers increasingly rely on machine learning tools to detect software vulnerabilities before release; however, the lack of high-quality training data often hinders their efforts. Addressing this, they aimed to develop an approach that generalises across similar projects by learning high-quality features. They employed a serialised AST and Continuous Bag of Words (CBOW) neural embedding to capture code semantics. These were processed using a deep learning algorithm to generate representations aligned with software vulnerabilities. This neural representation, derived from existing software projects, was then transferable to new projects, facilitating vulnerability detection even with limited training data. To evaluate their model, the researchers manually labelled 457 vulnerable functions and sourced over 30,000 non-vulnerable methods from six open-source projects. Their results indicated that the model could accurately identify vulnerabilities and adapt to various projects, outperforming traditional software metrics in both in-project and cross-project scenarios.

Russell et al. [2018] developed a vulnerability detection tool in response to the increasing emergence of new vulnerabilities, which heighten the risk of cyberattacks. They collected millions of C/C++ function-level code samples from the SATE IV Juliet Test Suite, Debian Linux Distribution, and GitHub. Using a custom lexer, they generated features from the source code, focusing on capturing the most relevant tokens to reduce the overall token vocabulary size. They implemented a CNN that interpreted the lexed source code for vulnerability detection. They achieved a Precision-Recall Area Under the Curve (PR AUC) of 0.944, an Area Under the Receiver Operating Characteristic Curve (ROC AUC) of 0.954, a Matthews Correlation Coefficient of 0.698, and an F1 score of 0.840. The study affirmed the effectiveness of deep feature representation learning on source code for vulnerability detection, suggesting it is a promising research direction.

Li et al. [2019] addressed the pressing issue of software security by introducing a method to efficiently detect vulnerabilities amidst rising software complexity and cyber threats. The study proposed a lightweight-assisted vulnerability discovery method using deep neural networks (LAVDNN) designed to identify weak functions in large-scale open-source software. This approach involved extracting function names as semantic features from source code and employing deep neural networks to differentiate between weak and benign functions. The research elaborated on the construction of deep neural networks and evaluated their performance across different models. The study found that LAVDNN effectively detected vulnerabilities in C/C++ and Python programs, achieving high F2 scores from 0.91 to 0.915, demonstrating the method's precision. Moreover, the approach significantly reduced false positive rates and efficiently pinpointed functions that required further analysis. In conclusion, the paper endorsed LAVDNN as a valuable tool for aiding manual code audits and enhancing the efficiency of vulnerability detection, noting its minimal need for preprocessing and reduced human intervention. Significant benefits were also highlighted, including the method's adaptability to various programming languages and its ability to identify vulnerabilities.

Feng et al. [2020a] emphasised the importance of automatic vulnerability detection in source code, highlighting the role of AI and deep learning. They critiqued existing methods that treat source code as plain text for not fully utilising syntax structure, which leads to redundancy and potential data loss from truncation techniques used for variable-length data. To overcome these limitations, they proposed a novel data processing approach using ASTs to capture all syntax-related features, thereby reducing redundancy. Their method involved parsing source code into ASTs to maintain syntax information and prevent data redundancy. They employed a pack-padded approach to handle variable-length data without truncation or padding. The model was evaluated on over 260,000 functions covering 118 CWE vulnerability types from the Juliet Test Suite in the National Institute of Standards and Technology (NIST) Software Assurance Reference Dataset (SARD). They reported F1 scores of 82.43 for buffer overflow vulnerabilities (CWE-121), 82.79 for types with over 5,000 samples ('5k'), and 82.76 for the entire dataset ('ALL'). The study concluded that while their approach significantly improved handling syntax-related features in vulnerability detection, the challenges necessitate further investigation.

Wang et al. [2021b] observed that despite ongoing efforts in vulnerability detection, the number of reported vulnerabilities continues to increase annually. They stated that traditional methods transform source code into an intermediate representation for machine learning or deep learning analysis. Still, this approach often leads to high false positive and false negative rates due to the representation's inability to accurately capture vulnerability characteristics. To that end, they introduced a feature extraction model using CNN to improve vulnerability detection. They analysed 65,513 code samples from the National Vulnerability Database and the NIST Software Assurance Reference Dataset. They calculated software metrics and derived secondary metrics to create a comprehensive dataset of code metrics. These samples were classified as 'vulnerable' or 'clean', incorporating various program size and complexity metrics. The study effectively utilised this enriched dataset in its CNN model, achieving precision and recall rates of approximately 80%. The researchers highlighted the need for further research into characterising source code to extract more comprehensive, high-quality features.

Dam et al. [2021] acknowledged that vulnerabilities and code flaws in software systems can lead to severe issues like deadlocking, hacking, data loss, and system failures. They added that traditional vulnerability detection methods rely on manually crafted features, such as complexity metrics or code token frequencies, which may not fully capture the necessary semantic and syntactic information for an effective prediction model. To address this, the researchers introduced an approach using the LSTM algorithm better to capture the semantic and syntactic attributes of source code. They tested this method on an Android dataset comprising 18 applications and a Firefox dataset, parsing source code into tokens and employing a 'codebook' concept from computer vision to develop features for both within-project and cross-project predictions. Their evaluation compared this approach against traditional software metrics, Bag of Words, and Deep Belief Network using four classifiers: Random Forests, Decision Tree, Naïve Bayes, and Logistic Regression. The results showed that all metrics (precision, recall, F-measure, and AUC) exceeded 80% for the Android dataset, and similarly high results were observed for cross-project evaluations. The study concluded that the researchers planned to extend their approach to other application types and programming languages, aiming to develop a comprehensive vulnerability prediction system that efficiently processes raw input to predict vulnerabilities.

Addressing the challenge of detecting software vulnerabilities before exploitation, Ziems and Wu [2021] introduced a method using Natural Language Processing (NLP) techniques to analyse code. This approach treats source code as text representations, utilising a database of over 100,000 C programming files identified with 123 vulnerabilities from the NIST Software Assurance Reference Dataset. They implemented five deep learning models, including LSTM, Bidirectional LSTM, and Bidirectional Encoder Representations from Transformers (BERT) The BERT models incorporate a transfer learning component trained in natural English. The hybrid model, which combines BERT and LSTM, achieved an accuracy of 93.19%, while the combination of BERT and BiLSTM recorded a slightly higher accuracy of 93.49%. The study highlighted the effectiveness of maintaining contextual information in NLP-based methods for vulnerability detection, suggesting that these techniques can significantly enhance the identification of potential security breaches in software.

Zou et al. [2021] acknowledged that granular vulnerability detection is crucial yet challenging. They stated that an ideal solution needs to detect vulnerabilities and identify their types. They pointed out that existing deep learning solutions can detect vulnerabilities but often fail to specify their types, a critical shortcoming given the need to quickly pinpoint vulnerabilities in large source files. The researchers introduced µVulDeePecker, a multiclass vulnerability detection system utilising deep learning, which addresses this gap by incorporating the 'Code Attention' concept. This concept builds upon the 'Code Gadget' idea to enhance the capture of semantic details in program analysis. This system comprises three main modules: a parser that transforms programs into Code Gadgets and Code Attentions, a vector representation extractor, and a detector module that utilises the LSTM algorithm for multiclass vulnerability detection. µVulDeePecker demonstrated impressive accuracy, achieving a false-positive rate of only 0.02%, a false negative rate of 5.73%, and an F1 measure of 94.22%. The research concluded that despite its success, the ongoing challenges in accurately classifying vulnerability types present significant opportunities for future research in vulnerability detection.

Zhuang et al. [2021] tackled the challenge of detecting software vulnerabilities, a crucial component of software security. The researchers stated that traditional methods, such as static and dynamic analyses, often suffer from high false positives and incomplete coverage, leading researchers to explore AI models for vulnerability detection. The study introduced a deep learning approach that automatically learns insecure patterns from code corpora, utilising a novel Graph Neural Network (GNN) architecture called 3GNN. The 3GNN model operates on disaggregated code graph representations, including ASTs, Data Flow Graphs (DFGs), and Control Flow Graphs (CFGs). It synthesises these representations and incorporates a new training loss metric that leverages the fine granularity of labelling. Tested on two real-world datasets, Draper and QEMU+Ffmpeg, 3GNN outperformed text, image, and graph-based approaches, achieving F1 scores between 0.54 and 0.62 and MCC scores between 0.25 and 0.52. The model's F1 score was 6.9% higher than that of the compared model, highlighting its effectiveness in capturing vulnerability signals. The study concluded with plans to explore the model's explainability and its application to new vulnerability detection datasets in future research.

Li et al. [2021] observed that most AI-driven vulnerability detection approaches only determine whether a piece of code is vulnerable without specifying which part of the code is at risk. To address this limitation, they developed IVDetect, an AI-driven tool to provide more detailed information on vulnerable statements. IVDetect consists of two main modules: a graph-based vulnerability detection model that takes the source code of a software system's methods as input and classifies them as vulnerable or non-vulnerable and a graph-based interpretation module that identifies and ranks the relevant vulnerable statements within these methods. The results showed significant improvements in top-10 nDCG and MAP ranking scores, ranging from 43% to 84% and 105% to 255%, respectively, compared to similar tools. IVDetect successfully identified the correct vulnerable statements in 7% of cases within a top-5 ranked list, with accuracy improvements ranging from 12.3% to 400% over other interpretation models. The study concluded by comparing IVDetect's performance with other AI-driven vulnerability detectors. The researchers planned to evaluate its performance against static analysis tools in future research.

Wartschinski et al. [2022] highlighted that identifying vulnerable code is crucial for enhancing software security, yet manual detection is time-consuming, requires expertise, and is inefficient. They asserted that automated vulnerability detection should meet critical criteria: high accuracy with granular identification, applicability across various software systems, and ease of use with minimal setup.

They developed VUDENC (Vulnerability Detection with Deep Learning on a Natural Codebase) to address these needs. This deep learning-based tool learns the features of vulnerable code from a real-world Python codebase. The researchers compiled a dataset of Python project commit histories from GitHub, focusing on seven OWASP Top 10 vulnerability types. Using the Word2Vec concept, they created vector representations of the code and trained their model with an LSTM algorithm. The model achieved recall rates of 78%–87%, precision rates of 82%–96%, and F1 scores of 80%–90%. The study emphasised the importance of understandability and actionability in making such tools practical for developers. The authors also suggested that future research could explore replacing their Word2Vec-based approach with more programming language-specific models, such as Code2Vec, which incorporate detailed AST-like features.

Binkley et al. [2022] argued that effective vulnerability prediction helps developers prioritise efforts by targeting the most at-risk software components. They critiqued Li et al. [2019]'s approach (also reviewed), which used deep learning to predict vulnerabilities based on function names, for its limited ability to identify and rank "dangerous" words. To address these shortcomings, the researchers proposed an improved method that systematically splits function names into constituent words, analysing the "danger" associated with each word. They defined a "dangerous" word as one whose presence in a function's name correlated with a higher likelihood of vulnerability. For instance, functions involving user input, which are often vulnerable to stack attacks, typically contain words like "read" or "input," which they classified as "dangerous." Functions with these terms were flagged as potentially vulnerable. The study analysed 73,000 vulnerable and 950,000 non-vulnerable functions, finding that the deep learning model heavily relied on individual words for classification, especially in datasets with a homogeneous vocabulary. The researchers concluded their approach could be efficient in within-project vulnerability prediction, where such vocabulary consistency is typical.

Hanif and Maffeis [2022] noted the steady rise in vulnerabilities since 2016, as reported by MITRE[2], which has driven increased research in vulnerability detection. They observed that deep learning techniques, particularly Bidirectional-LSTM and Graph Neural Networks, have shown promising results due to their ability to capture syntactic and semantic code information. Building on this success, the authors developed VulBerta, a transformer-based neural architecture. They pre-trained a RoBerta model using a custom tokenisation pipeline on functions from open-source C/C++ software projects. This deep knowledge representation, rich in syntactic and semantic information, was then used to train their vulnerability detection classifiers. A key aspect of their approach was its simplicity, which they attributed to achieving a slightly higher Matthews Correlation Coefficient of 55.86% compared to 52% in a related study by Zhuang et al. [2021] (also reviewed). The authors concluded that its conceptual simplicity and low complexity distinguish their research.

Luo et al. [2022] highlighted the significance of source code representation in AI-driven vulnerability detection. They introduced a programming language-agnostic technique called Compact Abstract Graphs (CAGs), designed for use with Graph Neural Network models. They derived CAGs from compact representations of abstract graphs built from the source code's ASTs. The performance of CAGs was evaluated on C and Java datasets from NVD and SARD. On the C dataset, they achieved a maximum accuracy of 93.21%, a precision of 92.51%, a recall of 93.47%, and an F1 score of 92.90%. For the Java dataset, the maximum values were 94.09% accuracy, 92.29% precision, 94.77% recall, and an F1 score of 93.40%. The study concluded that CAGs outperform traditional representations, such as ASTs, CFGs, and PDGs.

Zhang et al. [2022] highlighted the importance of vulnerability detection in information security, noting the limitations of existing approaches that rely on various code representation methods and deep learning algorithms. To overcome these challenges, they proposed a weight graph deep learning-driven approach. Their

[2]`https://www.cvedetails.com/browse-by-date.php`

method involved five key steps: data collection, static analysis, vector input, Weight Function Graph (WFG) transformation, and graph comparison, where the input is the source code and the output is identified as potential vulnerabilities. Using data from four programs, their tool, VDBWGDL, achieved a maximum F1 score of approximately 0.87. The study concluded that their approach outperformed existing vulnerability detection methods.

Fu and Tantithamthavorn [2022] highlighted the significant issues caused by vulnerabilities in software systems, such as deadlocks, information loss, and system failures, emphasising the importance of early vulnerability prediction during software development. They noted that IVDetect by Li et al. [2021] (also reviewed) and similar AI-driven approaches suffer from inaccuracies and lack precision. To address these shortcomings, they developed LineVul, designed to improve IVDetect by predicting vulnerable functions and pinpointing vulnerable lines. LineVul achieved a 91% F1 score in function-level prediction, surpassing IVDetect's performance. It also achieved a top-10 accuracy of 0.65 in line-level vulnerability localisation and a recall value of 0.75 for cost-effectiveness. The study concluded that LineVul is expected to assist security analysts in identifying vulnerable lines more cost-effectively.

Cheng et al. [2022] noted the increasing role of machine learning and deep learning across various fields, particularly in developing new static deep learning techniques for vulnerability detection as alternatives to traditional methods. They explained that current methods often abstract source code into graphs to train classification models that distinguish between vulnerable and non-vulnerable code fragments. However, they criticised this approach for focusing on classification rather than understanding the underlying vulnerability semantics. They introduced ContraFlow, a contrastive value-flow embedding approach designed for precise static software vulnerability detection to address this limitation. ContraFlow involves both a training and prediction process. The training phase includes contrastive value flow embedding, value-flow path selection, and model training. It takes value-flow paths extracted from unlabelled source code as input and produces a trained model. The prediction phase then applies this model to unseen code artefacts for vulnerability detection. Their results showed that the ContraFlow-Method achieved an F1 score of 75.3%, while the ContraFlow-Slice reached 82.8%. The study concluded that their approach outperformed similar methods developed in related works.

Tang et al. [2023] emphasised the importance of identifying and addressing potential vulnerabilities to secure software systems. They noted that traditional static vulnerability detection methods rely heavily on developer expertise. Furthermore, they added that most deep learning approaches typically use a single sequence or graph embedding method, often overlooking the structured information in the source code. To address these limitations, they developed a deep learning-based approach called Combining Sequence and Graph embedding for Vulnerability Detection (CSGVD). This approach models function-level vulnerability detection as a graph binary classification task. The approach has two phases: feature extraction and neural network model development. They introduced the PE-Bl module using the CodeXGLUE dataset, which includes over 27,000 samples from two sizeable open-source C projects. This module uses sequence embedding to extract semantic information from a code's control flow graph as node embeddings. They then utilised CSGVD's graph neural networks to capture the structural information of the control flow graph. Combining these elements, they developed feature representations for their neural model. Their approach achieved a maximum precision of 65.49% and a recall of 58.99%. The study concluded with the researchers expressing interest in further exploring node and graph embedding in future research.

3.2.4 Systematic Literature Reviews on Vulnerability Prediction

This subsection provides a concise overview of the field's trends and directions, reviewing systematic literature reviews on vulnerability detection and prediction. By juxtaposing the findings in this subsection with the literature discussed earlier, we aim to validate our conclusions on the state of the art in vulnerability detection and prediction and suggest future research directions.

Ghaffarian and Shahriari [2017] tackled the pressing issue of software security vulnerabilities, emphasising their significant impact on computer security. The study reviewed the increasing use of machine learning and data mining techniques for vulnerability analysis, categorised the existing works, discussed their benefits and limitations, and identified gaps in the research. The authors emphasised the importance of feature engineering and the need for novel machine learning algorithms specifically designed for software vulnerability analysis. They concluded that the field remains underdeveloped and lacks standard benchmark datasets. They recommended that future research focus on developing robust features, designing new algorithms, and establishing standard benchmarks for evaluation and comparison.

Jin and Yu [2018] noted that as software systems grow, so do the vulnerabilities they contain, making detection increasingly crucial. The researchers reviewed machine learning-driven methods for vulnerability prediction, focusing on program representation and vectorisation techniques. They discussed four primary code representation approaches often used in vulnerability prediction: software metrics, language models, and tree and graph representations. The study highlighted that supervised learning methods, particularly CNN and Bidirectional Long Short-Term Memory (BiLSTM) networks, are the most effective and are likely to dominate future research. The study concluded that current methods often operate at a coarse-grained level, highlighting the need for granular vulnerability prediction approaches.

Lin et al. [2020b] reviewed vulnerability detection methods leveraging deep learning. They observed that the rapid expansion of open-source software projects has created vast data opportunities for machine learning. The study highlighted how recent advances in deep learning, particularly in Natural Language Processing, have enabled neural models to better understand and identify vulnerable code patterns. The researchers focused on evaluating how these deep learning methods utilise neural networks to comprehend code semantics and identify vulnerabilities. They also identified key challenges in the field and proposed several promising future research directions.

Croft et al. [2022] emphasised that software vulnerability prediction relies heavily on data. They added that despite its growing popularity in software engineering, data preparation challenges hinder the widespread adoption of this approach in the industry. To address this, they systematically reviewed 61 peer-reviewed papers focusing on data preparation techniques and challenges in vulnerability prediction. The study identified 16 key challenges, including data generalisability, accessibility, scarcity, label noise, and data noise. The authors recommended consolidating these findings into a comprehensive resource to enhance data preparation efforts in vulnerability prediction research.

Nazim et al. [2022] emphasised the rising number of software vulnerabilities driven by the proliferation of new applications, highlighting the need for advanced detection methods to counter cyberattacks. They noted that AI-driven vulnerability detection is becoming increasingly important as human efforts alone are insufficient to manage these threats. They conducted a systematic literature review on deep learning-based vulnerability prediction methods to gain a deeper understanding of the field. Using five scientific databases, they identified 303 studies published between 2017 and 2022 and filtered them down to the ten most relevant papers. The review highlighted fundamental source code representation techniques, including ASTs, code gadgets, code property graphs, lexed representations, and semantics-based vulnerability candidates. Convolutional and recurrent neural networks were noted as the most prevalent models. Popular datasets mentioned were NVD, SARD, Draper VDISC, REVEAL, and FFMPeg+Qemu. The study concluded by acknowledging challenges, particularly the nascent stage of deep learning-based vulnerability prediction and the numerous unexplored areas in the field.

3.2.5 Comparative Studies on Vulnerability Prediction

This subsection reviews recent comparative studies on vulnerability detection and prediction, aiming to validate their conclusions by comparing their findings with the literature discussed earlier. These studies evaluate the effectiveness of deep learning versus traditional machine learning, assess the performance of various detection and prediction methods, and examine the factors that influence these approaches.

Theisen and Williams [2020] noted that the widespread adoption of vulnerability prediction remains limited despite extensive research using various metrics. They highlighted the need for software developers to have clear insights into the predictive power, data, and resource requirements of these models, which is essential for informed project decision-making. They also added that compared to traditional bugs, the relative rarity of vulnerabilities further hampers broader acceptance. To aid in selecting the most predictive features, the researchers conducted a comparative study of vulnerability prediction models. They replicated four models at the source file level on Mozilla Firefox, analysing 28,750 source code files with 271 vulnerabilities using software metrics, text mining, and crash data. They then examined the impact of combining features from these models on prediction performance. By integrating features from three models, they improved their best model's F1 score from 0.20 to 0.28. Their findings revealed that crash data, the FanOut dependency metric, and the string 'nullptr' (from their text mining model) were among the most predictive features. The study highlighted the importance of developing new features with strong predictive capabilities and suggested exploring novel analytical approaches to address the challenge of vulnerability scarcity in software systems.

Zheng et al. [2020] highlighted that despite numerous studies on vulnerability detection, theirs was the first to identify and evaluate the influence of four key factors on detection performance. These four factors were dataset quality, classification models, vectorisation techniques, and function or variable name changes. They emphasised that dataset quality, classification models, and vectorisation techniques directly impact detection outcomes. Additionally, they noted that changes in function or variable names could indirectly affect detection features and performance. The researchers conducted a comparative study to assess these factors, utilising two datasets: the National Vulnerability Database and the NIST Software Assurance Reference Dataset, and examining three vulnerability types: CWE-119, CWE-399, and CWE-664. Their most significant finding was that deep learning models, particularly Bidirectional LSTM, outperformed traditional machine learning models. This result aligns with findings from Hanif and Maffeis [2022] (also reviewed) and Jin and Yu [2018] (also reviewed), which support the effectiveness of Bidirectional LSTM. The study concluded by identifying areas for improvement, such as incorporating more vulnerability types and conducting a deeper analysis of dataset attributes, as the two datasets used exhibited different characteristics that influenced the results. They also stressed the need for more accurate and stable evaluation models.

Mazuera-Rozo et al. [2021] discussed the increasing use of deep learning techniques in software engineering, particularly in the context of vulnerability detection. However, they pointed out that the evidence supporting the superiority of deep learning over traditional machine learning remains inconclusive. To explore this further, they conducted an empirical study comparing the effectiveness of deep learning models with traditional machine learning techniques. The study evaluated two deep learning models, CNN and RNN, against a shallow machine learning model, specifically a Random Forest classifier. The evaluation focused on binary and multi-class classification of vulnerabilities using three C/C++ datasets, which contained 1.8 million functions, of which 400,000 were vulnerable. Additionally, they established a baseline with Google Cloud Platform's AutoML. Their findings revealed that the traditional Random Forest algorithm performed competitively with the deep learning models. This outcome suggests that traditional machine learning classifiers still provide a robust baseline against more advanced deep learning techniques. The study concluded that achieving reliability in vulnerability detection remains challenging and that current methods still have considerable room for improvement.

This section reviewed several vulnerability prediction studies, including those using supervised, semi-supervised, unsupervised, and deep learning techniques. The studies highlighted the importance of feature engineering, data preparation, and model selection in vulnerability prediction. They also identified key challenges, including data-related issues, various feature engineering approaches, and model performance, and suggested future research directions to address these challenges. In the following section, we review studies on general software bug prediction, which encompasses a broader perspective on vulnerabilities.

3.3 Software Bug Prediction

This section reviews studies on software bug prediction, focusing on bug prediction studies that influenced our work, machine learning-based bug prediction studies, and related topics. Since bugs are a superset of vulnerabilities, the studies reviewed in this section do not necessarily address bugs with security implications, like those in the preceding section.

3.3.1 Bug Prediction Studies Influencing Work

This subsection reviews studies that have significantly influenced our work, specifically in terms of our method-level granularity choice to vulnerability prediction.

Giger et al. [2012] stated that predicting bugs at a higher granularity than the method level is challenging, as it requires developers to examine thousands of lines of code to find predicted bugs. At the same time, extensive source code files are also more likely to be bug-prone. They proposed a method-level bug prediction technique to reduce developers' manual efforts and increase bug elimination efficiency. They utilised code change metrics and source code metrics from 21 open-source Java projects to develop bug prediction models, achieving a maximum precision of 84% and a recall of 88%. The study found that code change metrics significantly outperformed source code metrics. The researchers expressed interest in further exploring time-based code changes and expanding features for future prediction models.

Pascarella et al. [2018] pointed out that bug prediction aims to identify potentially defective software components, adding that researchers have found that combining product and process metrics yields the best results. In re-evaluating method-level bug prediction, they replicated Giger et al. [2012]'s study (also reviewed) across different systems and periods. They developed a method-level bug prediction model using the same features as Giger et al. [2012] and assessed its performance on 13 projects using 10-fold cross-validation. Notably, they did not use the SZZ algorithm, a popular algorithm for identifying bug-inducing changes, citing concerns about its reliability. Their results indicated that the models under-performed under rigorous evaluation, highlighting persistent challenges in method-level bug prediction.

3.3.2 Other Bug Prediction Studies

This subsection reviews other notable studies on bug prediction that have contributed to the advancement of the broader software bug prediction field.

Kim et al. [2008] asserted that the software development process would benefit significantly if developers could quickly identify defective changes. The researchers introduced change classification for bug prediction to address this need. They highlighted its advantages: granular prediction, no need for semantic source code information, applicability across languages and projects, and fast performance. They identified bug-introducing changes in 12 open-source projects, extracting features from metadata, change logs, source code, file names, and complexity metrics. Their classifier, which distinguishes between buggy and clean changes, achieved an accuracy of 78% and a recall of 60%. The researchers emphasised the importance of real-time assessment tools within IDEs and highlighted several unresolved challenges in this field.

Ferzund et al. [2009] asserted that managing software changes is challenging yet essential, as changes can introduce errors that lead to failures. They pointed out that these changes often occur in small code units and hunks across several source files. The researchers proposed a hunk classification technique using hunk-related metrics for granular bug prediction. They introduced various hunk metrics, processed revision histories to extract hunks, and identified bug-inducing ones. Using Logistic Regression and Random Forest algorithms, they developed models to classify hunks as either ‘buggy’ or ‘not buggy’ across seven open-source projects, achieving an accuracy of up to 81%, a precision of 77%, and a recall of 67%. The study also highlighted the varying effectiveness of individual metrics and suggested future research to refine these metrics further and explore different machine learning techniques.

Shivaji et al. [2009] stated that machine learning classifiers are widely used to predict buggy changes in source files. However, they often suffer from performance issues due to the inclusion of multiple features, which can slow down prediction times and hinder practical application. The researchers explored feature selection techniques in classification-based bug prediction to enhance performance by removing less essential features. They first identified bug fixes using log messages across 11 software projects and identified the related bug-inducing changes. After streamlining the features, they trained Naïve Bayes and Support Vector Machines classifiers with the reduced feature set. The study found that the Naïve Bayes classifier improved by 21% compared to Kim et al. [2008] (also reviewed), while the SVM classifier showed a 9% increase in F-measure. The researchers suggested that these improvements could facilitate the real-world adoption of classifier-based bug prediction by optimising performance and precision.

Yamada and Mizuno [2014] stated that many studies have focused on identifying and mitigating fault-prone software modules. They proposed a text filtering-based fault detection technique, hypothesising that bugs are related to specific words and contexts within a component. They used Git to obtain bug-fixing changes from two projects, Apache OpenJPA and Apache MINA and identified related bug-inducing changes. A text-filtering technique was then used to classify these changes. Their predictions for fault-prone modules in Apache OpenJPA achieved a recall of 0.97, a precision of 0.42, and an F1 measure of 0.60. For Apache MINA, the recall was 0.99, with a precision of 0.47 and an F1 measure of 0.64. The researchers expressed interest in testing their technique on more projects to compare its effectiveness.

An and Khomh [2015] stated that organisations are cautious about software crashes, often relying on automatic crash reporting tools to triage crash types and related bugs. They added that while these tools improve debugging efficiency, they act reactively after a crash occurs. To that end, the authors sought a more proactive approach by studying crash-inducing commits in Mozilla Firefox. They linked Firefox's crash reports to associated bugs and mapped these to relevant commits. From commit logs and source files, they extracted 24 metrics to develop predictive models for crash likelihood. Using algorithms such as GLM, Naïve Bayes, C5.0, and Random Forest, the classifiers achieved a precision of up to 61.4% and a recall of 95.0%. The study found that crash-inducing commits account for over 25% of all Firefox commits, often involving novice developers and significant code changes.

Ray et al. [2016] posited that real-life code resembles natural language, being repetitive and predictable. They suggested that various tools, such as suggestion engines and coding standard checkers, exploit the 'naturalness of software.' Code that seems improbable or 'surprising' to these models may be considered 'unnatural' and potentially faulty. The researchers analysed bug-fixing commits from 10 Java projects and traced corresponding bug-inducing commits using tools such as 'git-diff' and 'git-blame.' They assessed the naturalness of buggy code and its fixes, finding that it is typically more unnatural or entropic but becomes more natural after fixes. The study concluded that code entropy scores are a valuable indicator for defect prediction.

Despite extensive research in bug prediction, Bowes et al. [2016] noted that exploiting mutation testing by-products was still underexplored. They proposed a novel 'mutation-aware' fault prediction approach that combines traditional software metrics with mutation testing-related metrics to enhance bug prediction models. The effectiveness of this technique relies on the test suite's capability to detect bugs. They gathered mutation and traditional metrics from three substantial open-source and closed-source systems, encompassing over 220,000 lines of code. They applied Naïve Bayes, Logistic Regression, J48, and Random Forest models, cross-validated their approach, and evaluated them using the Matthews Correlation Coefficient. Their results demonstrated an improvement, with static code metrics achieving an MCC of 0.447, a 0.035 increase over similar studies. The combination of static and dynamic mutation metrics provided the best prediction performance, suggesting significant potential benefits for bug prediction and mutation testing fields.

Moussa et al. [2022] highlighted that software defects pose a significant challenge in the software industry, necessitating effective prediction methods to enhance software reliability. They added that traditional defect prediction models, which frame the task as a two-class classification problem, often suffer from performance issues due to the imbalance between defective and non-defective instances. Then, they pointed out that recent studies have considered one-class classifiers, such as the One-Class Support Vector Machines (OCSVM), which trains using only non-defective instances. To this end, they explored the efficacy of OCSVM across various defect prediction scenarios, including within-project, cross-version, and cross-project contexts. Then, they compared its performance with that of traditional classifiers such as Random Forest and Support Vector Machines. The study utilised empirical data from NASA and realistic datasets to evaluate OCSVM in three scenarios against classifiers such as Random Forest, Naïve Bayes, and Logistic Regression. Their evaluation metrics included the Matthews Correlation Coefficient and statistical tests such as the Wilcoxon Signed-Rank Test and Vargha and Delaney's A_{12} effect size. Their results showed that OCSVM often outperformed the Random Forest classifier in some cases, particularly in cross-version and cross-project predictions. Interestingly, they found that OCSVM sometimes surpassed Support Vector Machines, especially with heterogeneous data. A hyper-parameter-tuned version of OCSVM ($OCSVM_T$) improved performance, suggesting its suitability for diverse datasets. Their findings also showed that while OCSVM showed potential in defect prediction, especially when defective instances are scarce, Random Forest remained the most reliable classifier across scenarios. The study suggested further refining one-class classifiers and exploring their broader application to maximise their utility in defect prediction.

Shailee et al. [2024] emphasised the significance of tools for early defect detection in contemporary software systems. They utilised the NASA-curated JM1 dataset to evaluate various machine learning algorithms for predicting software bugs, aiming to enhance software quality and reduce maintenance costs. The study evaluated the effectiveness of various algorithms, including Naïve Bayes, Decision Trees, Random Forest, Support Vector Machines, Logistic Regression, Artificial Neural Networks, and K-Nearest Neighbors, using the JM1 dataset, which comprises 10,885 instances and 22 attributes. The dataset was divided into an 80%

training set and a 20% testing set. The researchers utilised Python-based machine learning libraries and evaluated the performance of their algorithms using precision, recall, F1 score, accuracy, and Root Mean Squared Error (RMSE) metrics. Random Forest emerged as the top performer, achieving an accuracy of 81%, the highest precision and recall rates, and the lowest RMSE, indicating its effectiveness in predicting software defects. Logistic Regression also performed well, with an accuracy of 80%. Other models exhibited varied performance levels but were generally less effective. The results confirm that Random Forest is an effective tool for early bug detection, suggesting its potential for broader application in software development. The study recommends further research on advanced feature engineering and applying these models to different datasets and software projects to enhance their generalisability and effectiveness.

Chowdhury et al. [2024] addressed the challenges posed by software bugs, noting the limited practical adoption of existing bug prediction models due to their coarse granularity at class or file levels. Their study focused on method-level bug prediction (MLBP), aiming to provide more precise and actionable insights for the early detection and resolution of bugs. The research evaluated the efficacy of MLBP models in realistic, time-sensitive scenarios to identify limitations and propose improvements. They used three publicly available datasets and a new dataset comprising 774,051 Java methods from 49 open-source projects. The study assessed MLBP models under realistic conditions, rather than using traditional k-fold cross-validation, to prevent incorporating future data into the training process. They introduced a more accurate bug labelling approach to reduce noise, investigated method age as a predictor for concept drift, and explored the effectiveness of selecting optimal training projects. The research also evaluated the benefits of using separate models for small and large methods. Their findings indicated that existing MLBP models underperform in realistic scenarios due to issues such as noisy data and the misuse of future information during training. The study showed improved prediction accuracy with a refined bug labelling approach, confirmed that method age could effectively capture concept drift, and highlighted that tailored models for different method sizes could enhance accuracy. The study concluded that while MLBP is an open research area, addressing issues such as data noise, concept drift, and project selection for training could significantly enhance the models' practical applicability. The study recommended that future research focus on refining labelling techniques, leveraging method age, and developing tailored models to enhance MLBP effectiveness in software engineering.

This section concludes our literature review, summarising the key findings from vulnerability and bug prediction studies. The subsequent section will present our observations from the reviewed literature, highlighting trends, challenges, and future research directions in the field.

3.4 Observations in the Literature

This section summarises our observations from the reviewed literature, covering datasets, methodologies, evaluation metrics, outcomes, and emerging research trends.

3.4.1 Dominance of Deep Learning Techniques

Deep learning techniques have dominated vulnerability prediction research for the better part of the last decade. However, Mazuera-Rozo et al. [2021] challenges the assumed superiority of deep learning, demonstrating that traditional machine learning can achieve comparable results. Regardless, deep learning remains favoured, possibly due to trends, perceived sophistication, or technical familiarity.

3.4.2 Success Stories of the Long Short-Term Memory Algorithm

Recent studies consistently highlight the LSTM algorithm and its variants as the leading methods for vulnerability prediction based on code representation.

3.4.3 Random Forest as a Reliable Baseline

Random Forest has emerged as a reliable baseline for both vulnerability and bug prediction, often consistently outperforming many other traditional machine learning algorithms. This performance could be attributed to the Random Forest classifier's ability to handle high-dimensional data and complex relationships between features.

3.4.4 Challenges in Adopting Vulnerability Prediction

Vulnerability prediction faces considerable barriers to widespread adoption, with several factors hindering broader implementation. These challenges include technical issues such as data quality and availability, the complexity of modern software systems, and the absence of standardisation. We briefly explore these factors below.

3.4.4.1 Data Challenges

A key challenge in developing effective vulnerability prediction models is the limited availability of large, diverse, and representative datasets. Accurate models rely on comprehensive data about past vulnerabilities, their characteristics, and corresponding fixes. However, such datasets are often scarce or inaccessible, complicating the training and evaluation of prediction models.

3.4.4.2 Complexity of Modern Software Systems

Modern software systems are inherently complex, with numerous dependencies, libraries, and components. This complexity makes it challenging to develop a vulnerability prediction solution that effectively addresses these diverse elements. Vulnerabilities may emerge from interactions between components or unexpected behaviours between software layers. Additionally, the continuous evolution of software, with frequent updates, patches, and new features, further complicates prediction efforts.

3.4.4.3 Lack of Standardisation

A standardised approach is crucial for the widespread adoption of AI-driven vulnerability prediction. While there is consensus in the literature on the rapid growth of software vulnerabilities and the relevance of AI techniques, there is significant disharmony on the steps required to implement practical solutions. This lack of standardisation hinders software professionals from adopting these methods outside the research community.

3.5 Future Research Directions

This section outlines future research directions based on the reviewed literature. Based on the identified challenges and opportunities, we provide recommendations for advancing vulnerability prediction research and highlight areas for further exploration.

3.5.1 Real-Time Prediction

As Kim et al. [2008] suggested, integrating real-time bug prediction tools within IDEs could greatly benefit developers. Future research should explore more efficient and seamless integration methods.

3.5.2 Granular Prediction and Contextual Information

Research by Giger et al. [2012] and Ferzund et al. [2009] has shown the benefits of method-level and hunk-level prediction. Future studies should continue to explore these granular techniques to enhance their performance.

3.5.3 Leveraging Large Language Models (LLMs) for Vulnerability Prediction

With the rise of Large Language Models, future research should explore leveraging these models for vulnerability prediction.

3.5.4 Data Preparation and Standardisation

Data-related challenges and the lack of standardisation are major obstacles in vulnerability prediction research. Future work should aim to develop standardised datasets and data preparation techniques to improve model training and evaluation. Creating benchmark datasets that cover various vulnerabilities, software systems, and programming languages is crucial for advancing the field. Additionally, developing comprehensive datasets that mirror real-world software development conditions is essential. Future research should prioritise creating and using realistic datasets for more accurate evaluation of vulnerability prediction models.

3.6 Conclusion

We have reviewed the vulnerability and bug prediction literature, summarising key findings and offering insights for future research. Our review highlights the growing focus on developing more effective prediction models, mainly through AI-driven techniques. This shift has spurred innovation in feature engineering, which is now a central focus in the field. Notable techniques include the codebook method by Dam et al. [2021], the mutation-aware approach by Bowes et al. [2016], and the concept of the 'naturalness of software' by Ray et al. [2016]. We contribute to this area by proposing a novel feature engineering technique that uses information retrieval methods to build features encoding vulnerability code semantics from known vulnerable code samples. This technique represents a novel contribution to the field of vulnerability prediction. To our knowledge, it is the first information retrieval-driven technique in the field of vulnerability prediction research. We are confident that it will significantly advance vulnerability prediction and aid researchers in developing more effective models, whether from scratch or as a data augmentation technique for existing models, as well as for the newly emerging Large Language Models. In the following two chapters, we will present our proposed feature engineering technique and evaluate its effectiveness in predicting vulnerabilities in real-world software systems.

Chapter 4

Token-Based Vulnerability Prediction

This chapter introduces a novel Information Retrieval-driven technique for predicting software vulnerability. It uses token-based source code representations to develop novel predictive software metrics. The chapter presents the methodology, results, and discussion of the technique's performance in a Within-Project setting, addressing the first research question of this thesis.

4.1 Introduction

As software systems have become increasingly integral to our daily lives, their complexity and interconnectedness have also grown. These systems typically comprise multiple software components that communicate with each other and other systems over a network. Unfortunately, this complexity and interconnectivity create a fertile ground for bugs and vulnerabilities. Gujral et al. [2015] described a bug as a software flaw that causes the system to deviate from its specification. Al Debeyan et al. [2022] defined a vulnerability as a weakness in a software system that can be exploited to compromise its security. Thus, vulnerabilities are a subset of bugs characterised by their security implications.

The increasing complexity of software systems makes identifying vulnerabilities more challenging, necessitating the development of advanced techniques and tools. These new methods must go beyond traditional approaches to identify vulnerabilities early in the software development lifecycle. In this chapter, we detail our novel technique for identifying vulnerabilities. Our approach leverages information retrieval-driven techniques to enhance the effectiveness of vulnerability prediction.

4.1.1 Chapter Motivation

The need for advanced techniques to identify vulnerabilities in software systems has spurred numerous studies proposing innovative prediction methods. However, many of these approaches fail to yield satisfactory results.

These methods face two main issues. Firstly, they often lack the performance required for practical use. Secondly, they typically operate at a granularity level that is too coarse, making them impractical. This concern is noted by Morrison et al. [2015] and Al Debeyan et al. [2022]. By granularity, we mean the unit of analysis within the software system. For instance, a class-level prediction approach identifies vulnerable classes in the software system, and a method-level prediction approach identifies vulnerable methods.

4.1.2 Research Question

The growing interest in vulnerability prediction has led to numerous studies proposing innovative approaches with varying degrees of success. Researchers have applied diverse concepts from other fields of computer science to vulnerability prediction. However, significant unexplored areas remain, particularly the application of information retrieval techniques to vulnerability prediction. We address this gap by proposing an information retrieval-driven software vulnerability prediction technique.

Information retrieval involves finding relevant information from extensive data collections to help users locate the information they need quickly and efficiently. Information retrieval techniques are widely utilised in search engines, digital libraries, and data mining applications. They primarily focus on text-based data but can also operate on images, audio, and video. The information retrieval process typically includes document collection, indexing, query processing, ranking, and information presentation [Schütze et al., 2008, Salton, 1983].

Pattern matching, a core characteristic of information retrieval, involves identifying similarities between a query and documents within a collection. This capability can be leveraged to encode patterns indicative of software vulnerabilities, aiding in the development of security-focused predictive metrics for machine learning-based vulnerability prediction.

In this chapter, we address the following research question:

> *How well does the information retrieval-driven software vulnerability prediction technique perform on a single, multi-release software system dataset for token-based source code representations?*

4.1.3 Research Scope

This study focuses on the following areas:

- **Programming Language:** The datasets used are written in Java, and the vulnerabilities are within Java methods. Vulnerabilities from other sources, such as web services, annotations, and configuration files, are not taken into consideration.

- **Method-Level Vulnerability Prediction:** The focus is on predicting vulnerabilities at the method level rather than at the class or file level.
- **Within-Project Vulnerability Prediction:** The study predicts vulnerabilities within a single software system across multiple releases.
- **Binary Classification:** The study employs binary classification using machine learning to predict whether a method is vulnerable without considering multi-class classification, i.e., predicting the type of vulnerability.

4.1.4 Significance and Contributions

This chapter proposes an information retrieval-based technique for developing software metrics that predict vulnerabilities, leveraging the pattern-matching capabilities of information retrieval to construct security-relevant metrics from source code structures.

4.1.4.1 Significance of the Study

The study demonstrates the feasibility of repurposing information retrieval techniques for vulnerability prediction. These techniques effectively develop security-relevant predictive software metrics, potentially increasing their adoption for efficiently identifying and mitigating software vulnerabilities.

4.1.4.2 Contributions

This study introduces several novel information retrieval-driven software metrics for vulnerability prediction.

4.1.5 Structure of the Chapter

The rest of this chapter is structured as follows: Section 4.2 provides the background for this chapter. Section 4.3 introduces our new information retrieval-driven software metrics for vulnerability prediction. Section 4.4 outlines the methodology used in this chapter. Section 4.5 presents the chapter's results. Section 4.6 discusses the findings and their implications. Section 4.7 addresses the threats to the chapter's validity. Finally, Section 4.8 answers the first research question of this thesis.

4.2 Background

The rapid adoption of technology has brought significant privacy and security challenges [Al Debeyan et al., 2022]. Software vulnerabilities are security weaknesses in code that can be exploited to gain unauthorised control of a system, steal or manipulate sensitive data, or deny system services [Liu et al., 2019]. Detecting, mitigating, and eradicating vulnerabilities are critical for software security and reliability [Riom et al., 2021]. Identifying vulnerable components early in development enables developers to prioritise efforts on critical areas, thereby improving security, reducing remediation costs, and shortening the time-to-market for software products.

Various approaches exist for identifying vulnerabilities, including penetration testing, static code analysis, and dynamic analysis. Modern vulnerability prediction, utilising machine learning, deep learning, and data mining, aims to identify vulnerabilities before they are exploited, addressing the limitations of traditional techniques [Ghaffarian and Shahriari, 2017]. These methods typically involve classification models to predict vulnerable code areas, enabling developers to target their security auditing efforts more effectively. However, current methods often underperform and offer coarse-grained predictions [Chakraborty et al., 2021, Al Debeyan et al., 2022].

Several factors influence the performance of vulnerability prediction models, with code representation being a crucial factor. Previous models have used static code metrics [Shin and Williams, 2008b, Sultana et al., 2021], literal text tokens [Zou et al., 2021, Scandariato et al., 2014], and graph representations such as Control Flow Graphs, Data Dependency Graphs, or Abstract Syntax Trees (ASTs) [Bilgin et al., 2020, Cao et al., 2021, Liu et al., 2019, Partenza et al., 2021, Zhou et al., 2019b]. These representations serve as input features for machine learning or deep learning models.

Contemporary approaches transform source code into a format compatible with machine learning or deep learning algorithms, often numerical. This chapter uses token-based source code representation and shingling to develop novel information retrieval-driven metrics for vulnerability prediction. Background on source code representation, particularly token-based, is provided in Section 2.3 and Subsection 2.3.1. However, given its central role in this thesis, we briefly reiterate the relevant techniques in the following subsection.

4.2.1 Token-Based Source Code Representation

As discussed in Section 2.3, source code representation is vital in modern software engineering, including vulnerability prediction. It transforms text-based source code into a more abstract, structured form, effectively retaining syntactical and semantic information [Zhang et al., 2019]. This abstraction aids various tasks, such as program analysis, transformation, and optimisation, by converting source code into formats suitable for machine learning and deep learning algorithms [Hancock and Khoshgoftaar, 2020].

Source code representations can be token-based, tree-based, or graph-based [Samoaa et al., 2022]. These approaches abstract away low-level details, focusing instead on high-level code structures, making complex tasks more manageable. A common technique, tokenisation, breaks down text into smaller units called tokens, such as keywords, identifiers, literals, and operators.

Shingling, another related concept, breaks text into overlapping subsequences called shingles, capturing local text structure. For example, the sentence *'The quick brown fox jumps over the lazy dog'* yields the following 3-gram shingles: *'The quick brown', 'quick brown fox', 'brown fox jumps', 'fox jumps over', 'jumps over the', 'over the lazy', 'the lazy dog'*. Shingling can help capture the context of code elements in a method.

Our contributions to vulnerability prediction research involve leveraging token-based source code representation and shingling to develop novel information retrieval-driven metrics. These metrics translate the structure of methods into a numerical form that machine learning algorithms can understand. Some of our developed metrics capture method attributes directly, while others use patterns consistent with code vulnerabilities to enhance predictive power.

We focus on method-level granularity, predicting vulnerabilities within methods of a software system. Granularity refers to the level of detail in vulnerability predictions [Lomio et al., 2022], and many researchers have highlighted its importance [Al Debeyan et al., 2022, Morrison et al., 2015]. We had previously discussed granularity levels in Subsection 2.2.1. However, in the following subsection, we reiterate the importance of method-level granularity to set the context for understanding our metrics.

4.2.2 Granularity Levels in Software Vulnerability Prediction

The most common granularity levels in modern object-oriented programming languages are binary, source code file, class, and method. For example, a class-level vulnerability prediction solution identifies vulnerable classes, while a method-level solution identifies vulnerable methods within a software system.

Among these, binary level granularity is the coarsest, and method level is the finest. Additionally, recent studies have explored code-change level granularity, which is even more granular than method level [Şahin et al., 2022, Giger et al., 2011, Shivaji et al., 2012, Yang et al., 2023]. However, as Russell et al. [2018] noted, method-level granularity captures a subroutine's flow more completely, providing better context than code-change level granularity.

Identifying vulnerabilities in binary or source code files can be relatively straightforward. However, this coarse granularity requires developers and security testers to invest significant time and effort in pinpointing the exact location of the vulnerability, which may not be immediately apparent within a significant software component [Morrison et al., 2015].

Method-level vulnerability prediction involves identifying potentially vulnerable methods within a software system. Unlike coarser granularities, such as binary and source code file levels, analysing individual methods can be more complex. However, remediating vulnerabilities at the method level is usually more straightforward once identified due to the methods' smaller size compared to classes and source code files, which reduces the effort required for inspection.

Our study focuses on method-level vulnerability prediction. We use token-based source code representation and shingling to develop novel information retrieval-driven software metrics. Therefore, the metrics introduced in the next section have a method-level scope.

4.3 Token-Based Software Metrics

This section highlights the novelty of our study. We developed sixteen custom software metrics, driven by information retrieval, that leverage token representations in our datasets. These metrics are used as machine learning classification features to predict the vulnerability proneness of software components. Our datasets include a target software system and a vulnerability dataset. The target software system is the software for which we predict vulnerabilities, while the vulnerability dataset contains known software vulnerabilities used to develop specific metrics.

Vulnerable and non-vulnerable software artefacts often have similar attributes, which can complicate the differentiation process [Pereira et al., 2021], particularly in machine learning-based vulnerability prediction. So, to enhance predictive power in recognising patterns consistent with code vulnerabilities, we integrated thousands of known software vulnerabilities from the vulnerability dataset to develop metrics that capture patterns consistent with code vulnerabilities. These metrics are used to predict the vulnerability proneness of software components in the target software system.

The Methodology section will provide further details on our target software system and vulnerability dataset. Meanwhile, the rest of this section will elaborate on the metrics, categorised into hit-independent and hit-dependent metrics.

A ‘hit’ refers to code fragments in the target software system that match fragments in the vulnerability dataset, represented as shingles. In this chapter, *a hit refers to the intersection of shingles from a target software system method under consideration and those from a vulnerable method in the vulnerability dataset.*

For instance, if the shingles of a method in our vulnerability dataset share one or more shingles with the `printHelloWorld` method in Listing 4.1 whose token representation and shingles (separated by ‘$\leftrightarrow$’) are shown in Figures 4.1 and 4.2 respectively, we say that the `printHelloWorld` method has a hit with the vulnerability dataset method.

Listing 4.1: `printHelloWorld` Method

```java
    public void printHelloWorld() {
        System.out.println("Hello, World!");
    }
```

```
public void printHelloWorld ( ) { System . out . println
    ↪ ( " Hello , World ! " ) ; }
```

Figure 4.1: Token Representation of the Method in Listing 4.1

```
public void printHelloWorld ( )↔void printHelloWorld ( )
    ↪  {↔printHelloWorld ( ) { System↔( ) { System .↔)
    ↪ { System . out↔{ System . out .↔System . out .
    ↪ println↔. out . println (↔out . println ( "↔.
    ↪ println ( " Hello↔println ( " Hello ,↔( " Hello ,
    ↪ World↔" Hello , World !↔Hello , World ! "↔, World
    ↪  ! " )↔World ! " ) ;↔! " ) ; }
```

Figure 4.2: 5-gram Shingles of the Method in Listing 4.1

We use the term *hit-independent metrics* for metrics that are calculated based on concrete attributes (such as code churn, size, and complexity) discernible from a target software system method's token representation without relying on hits. Conversely, *hit-dependent metrics* refer to metrics that rely on the concept of *hit* for their calculation. The following subsections will provide more details on both hit-independent and hit-dependent metrics.

4.3.1 Token-Based Hit-Independent Metrics

Table 4.1: Token-Based Hit-Independent Metrics

Token-Based Hit-Independent Metric	**Abbr.**
Number of Target Software System Tokens	NTT
Number of Distinct Target Software System Tokens	NDTT
Token-Based Instantaneous Code Churn	TICC
Token-Based Relative Instantaneous Code Churn	TRICC
Number of Target Software System Diff Tokens	NTDT
Number of Target Software System Distinct Diff Tokens	NTDDT
Token Relative Uniqueness	TRU

Table 4.2: Token-Based Hit Independent Metrics Code Attributes of Concern

Token-Based Hit-Independent Metric Abbr.	**Attributes of Concern**		
	Churn	*Intricacy*	*Size*
NTT			✓
NDTT		✓	✓
TICC	✓		
TRICC	✓		
NTDT			✓
NTDDT		✓	✓
TRU		✓	

Table 4.1 presents the seven hit-independent metrics we developed. Table 4.2 shows the code attributes each metric captures.

4.3.1.1 Number of Target Software System Tokens (NTT)

A target software system method's NTT is the total number of tokens in its token representation. This straightforward metric indicates the size of a method. For example, the NTT of the method shown in Figure 4.1 is 20. *The hypothesis is that more extensive methods are more likely to contain vulnerabilities.*

For a target software system method m_t with a *multiset* of tokens $T_t(m_t)$, NTT is expressed as:

$$\text{NTT} = |T_t(m_t)|$$

The NTT metric's hypothesis was inspired by several studies, including Shin et al. [2010], Giger et al. [2012], Morrison et al. [2015], Pascarella et al. [2018], and Du et al. [2019]. However, we conceptualised its information retrieval-based design and implementation.

4.3.1.2 Number of Distinct Target Software System Tokens (NDTT)

A target software system method's NDTT is the count of *unique* tokens in its token representation. Unlike NTT, which indicates size, NDTT measures both the size and diversity of code elements. A higher NDTT suggests a more diverse and intricate method. For example, the NDTT of the method shown in Figure 4.1 is 16. *The hypothesis is that intricate methods are more likely to contain vulnerabilities.*

For a target software system method m_t with a *set* of tokens $T_t'(m_t)$, NDTT is expressed as:

$$\mathrm{NDTT} = |T_t'(m_t)|$$

The NDTT metric was inspired by the NTT metric, which in turn was inspired by several studies, including Shin et al. [2010], Giger et al. [2012], Morrison et al. [2015], Pascarella et al. [2018], and Du et al. [2019]. However, we conceptualised its information retrieval-based design and implementation.

4.3.1.3 Token-Based Instantaneous Code Churn (TICC)

A target software system method's TICC is the number of times its token representation has changed throughout its history. Code churn quantifies the frequency of code rewrites over time, measured by version control check-ins or changes in lines of code [Shin et al., 2010]. This metric is commonly used in vulnerability prediction, based on the hypothesis that higher churn correlates with higher vulnerability proneness [Zimmermann et al., 2010, Shin et al., 2010, Shin and Williams, 2013, Meneely et al., 2013, Morrison et al., 2015].

In our work, any code change, however minor, alters the token representation, so TICC is incremented for each release where the method's token representation changes. Thus, TICC counts the number of changes to a method's source code across the software system's history. *The hypothesis is that methods that undergo frequent modifications are more likely to contain vulnerabilities, possibly indicating problematic code.*

For a method m_t in a target software system, with $\delta(m_t)$ representing the set of releases where m_t's token representation changed, TICC is evaluated as:

$$\mathrm{TICC(m_t)} = |\delta(m_t)|$$

The TICC metric's hypothesis was inspired by several studies, including Shin et al. [2010], Giger et al. [2012], Morrison et al. [2015], Pascarella et al. [2018], and Du et al. [2019]. However, we conceptualised its information retrieval-based design and implementation.

4.3.1.4 Token-Based Relative Instantaneous Code Churn (TRICC)

A target software system method's TRICC is the ratio of its token representation change count to the total number of releases in which it appears. TRICC is similar to TICC, but it is a relative value that better indicates how frequently each method has evolved compared to others.

For a method m_t in a target software system, and N representing the total number of releases where m_t appears, TRICC is calculated as:

$$\mathrm{TRICC(m_t)} = \frac{TICC(m_t)}{N}$$

In a later section, we will discuss the design of the TRICC metric further, particularly how N could threaten the validity of a machine learning analysis through data leakage, depending on the context of the analysis.

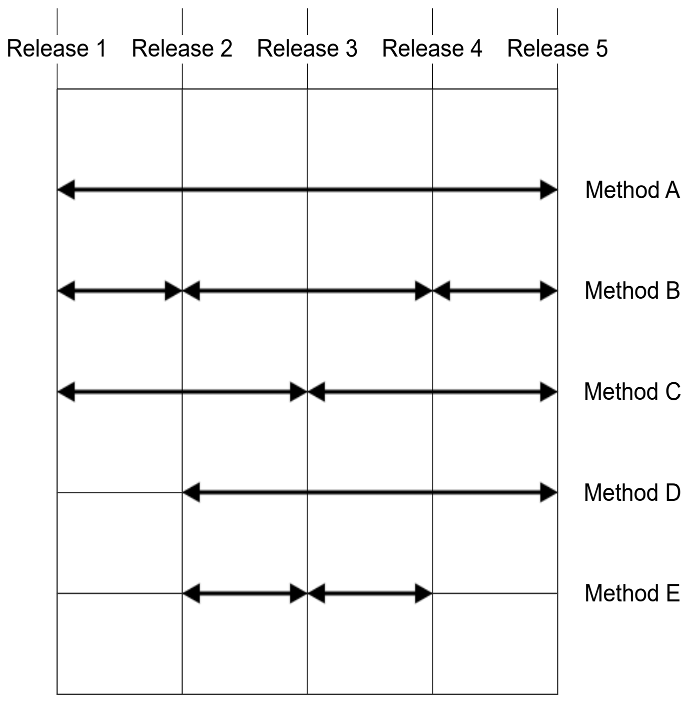


Figure 4.3: Method Evolution: An Infographic Representation

To illustrate TRICC, Figure 4.3 shows the evolution of methods A, B, C, D, and E over releases 1 to 5 of a target software system.

- Method A first appeared in Release 1 and remained unchanged.

- Method B first appeared in Release 1 and was modified in Releases 2 and 4.

- Method C first appeared in Release 1 and was modified only in Release 3.

- Method D first appeared in Release 2 and remained unchanged thereafter.

- Method E first appeared in Release 2, was modified in Release 3, and was removed in Release 4.

Table 4.3: TICC and TRICC Illustration

Method	Release 1	Release 2	Release 3	Release 4	Release 5
A	TICC:0/RICC:0	TICC:0/RICC:0	TICC:0/RICC:0	TICC:0/RICC:0	TICC:0/RICC:0
B	TICC:0/RICC:0	TICC:1/RICC:0.20	TICC:1/RICC:0.20	TICC:2/RICC:0.40	TICC:2/RICC:0.40
C	TICC:0/RICC:0	TICC:0/RICC:0	TICC:1/RICC:0.20	TICC:1/RICC:0.20	TICC:1/RICC:0.20
D	NA	TICC:0/RICC:0	TICC:0/RICC:0	TICC:0/RICC:0	TICC:0/RICC:0
E	NA	TICC:0/RICC:0	TICC:1/RICC:0.33	TICC:1/RICC:0.33	NA

Table 4.3 illustrates the TICC and TRICC metrics for the methods in Figure 4.3.

- **Method A**: First introduced in Release 1 and remained unchanged in subsequent releases. TICC and TRICC values are 0 throughout the five releases.

- **Method B**: First appeared in Release 1, modified in Release 2 and 4. TICC is 1 in Release 2 and 3, and 2 in Release 4 and 5. TRICC is 0.20 in Release 2 and 3, and 0.40 in Release 4 and 5.

- **Method C**: First appeared in Release 1, modified only in Release 3. TICC is 1 in Release 3, 4, and 5. TRICC is 0.20 in Releases 3, 4, and 5.

- **Method D**: First introduced in Release 2 and remained unchanged in subsequent releases. TICC and TRICC values are 0 throughout the four releases.

- **Method E**: First appeared in Release 2, modified in Release 3, and removed in Release 4. TICC is 1 in Release 3 and 4. TRICC is 0.33 in Release 3 and 4.

The TRICC metric was inspired by the TICC metric, which in turn was inspired by several studies, including Shin et al. [2010], Giger et al. [2012], Morrison et al. [2015], Pascarella et al. [2018], and Du et al. [2019]. However, we conceptualised its information retrieval-based design and implementation.

4.3.1.5 Number of Target Software System Diff Tokens (NTDT)

A target software system method's NTDT is the count of tokens in the symmetric difference between its tokens and those of its previous release. As discussed for TICC, any code changes alter the token representation. The NTDT metric measures the magnitude of changes between two contiguous releases by counting the tokens in the difference.

The hypothesis is that developers are more likely to introduce vulnerabilities when making significant and intricate changes to a method.

For a method m_t in a target software system, and m_{t-1} representing its previous release, NTDT is expressed as:

$$\mathrm{NTDT(m_t)} = |T_t(m_t)\,\Delta\,T_{t-1}(m_{t-1})|$$

Here, Δ is the symmetric difference operator that returns a *multiset* of elements in either of the two sets but not in both, and $T_t(m_t)$ and $T_{t-1}(m_{t-1})$ are the token representations of m_t and m_{t-1}, respectively.

The NTDT metric's hypothesis was inspired by several studies, including Shin et al. [2010], Giger et al. [2012], Morrison et al. [2015], Pascarella et al. [2018], and Du et al. [2019]. However, we conceptualised its information retrieval-based design and implementation.

4.3.1.6 Number of Target Software System Distinct Diff Tokens (NTDDT)

A target software system method's NTDDT is the count of *unique* tokens in the symmetric difference between its tokens and those of its previous release. NTDDT is similar to NTDT, but while NTDT measures the magnitude of changes between two contiguous releases, NTDDT measures the diversity of the code elements involved in those changes.

For a method m_t and its previous release m_{t-1}, NTDDT is expressed as:

$$\mathrm{NTDDT(m_t)} = |T_t'(m_t)\,\Delta\,T_{t-1}'(m_{t-1})|$$

Here, Δ is the symmetric difference operator that returns a *set* of elements in either of the two sets but not in both, and $T'_t(m_t)$ and $T'_{t-1}(m_{t-1})$ represent the *set* of tokens in the token representations of m_t and m_{t-1}, respectively.

The NTDDT metric was inspired by the NTDT metric, which in turn was inspired by several studies, including Shin et al. [2010], Giger et al. [2012], Morrison et al. [2015], Pascarella et al. [2018], and Du et al. [2019]. However, we conceptualised its information retrieval-based design and implementation.

4.3.1.7 Token Relative Uniqueness (TRU)

The TRU metric analyses the individual tokens in a method's token representation. It is inspired by the Term Frequency-Inverse Document Frequency (TF-IDF) technique from information retrieval and is used for method-level vulnerability prediction. The idea is that within the context of a given software system or codebase, the developers working on it are more likely to write vulnerable code when using complex programming features that the team rarely uses. The TRU metric reflects a method's intricacy based on the rarity of its code elements compared to other methods in the software system or codebase.

TF-IDF measures a word's importance in a document relative to its rarity across a corpus. Similarly, TRU assesses the uniqueness of tokens within methods in a target software system.

In our context, a word corresponds to a token, a document to a method, and a corpus to the target software system. We apply the TF-IDF technique for method-level vulnerability prediction to assign weightings to each method's token, indicating how unique a token is compared to those in other methods.

To illustrate, given a software system, the Java keyword 'void' commonly appears in standard methods that do not return a value. In contrast, the keyword 'synchronized' is more likely to appear in intricate methods that handle threading. We will observe that in a synchronized void method, the 'synchronized' keyword will likely have a higher TF-IDF weighting than 'void' because 'void' is more frequently used across methods in the software system.

If we compare a void method and a synchronized void method of similar size and standard code elements, the synchronized void method will have a higher TRU. This suggests that methods containing the 'synchronized' keyword are more complex than average void methods, making them more prone to vulnerability-inducing changes.

The TRU metric indicates a method's uniqueness compared to other methods. It is calculated as the harmonic mean of all TF-IDF weightings for a method's tokens. Thus, TRU measures the relative obscurity of a method compared to others. A method with a high TRU value is likely to feature advanced, specialised, and rarely used programming language concepts in the software system. *The hypothesis is that developers working on a given software system or codebase are more likely to introduce vulnerabilities when working with advanced and complex programming language features that are rarely used by the development team.*

Suppose t represents a token in a target software system's method, and m represents a *multiset* of tokens in the method. The term frequency, $\mathrm{tf}(t,m)$, is the relative frequency of token t within the method m. It is expressed as:

$$\mathrm{tf}(t,m) = \frac{f_{t,m}}{\sum_{t' \in m} f_{t',m}}$$

Here, $f_{t,m}$ represents the raw count of token t in the method's representation, and $\sum_{t' \in m} f_{t',m}$ denotes the total number of tokens in m.

Suppose M represents a *multiset* of all method tokens in our target software system. We express the inverse document frequency, $\mathrm{idf}(t,M)$, for token t in method m as:

$$\mathrm{idf}(t,M) = \log \frac{N}{|m \in M : t \in m|}$$

Here, N is the total number of methods in the target software system, $N = |M|$, and $|m \in M : t \in m|$ is the total number of methods that include t.

The TF-IDF for token t is then calculated as:

$$\mathrm{tfidf}(t,m,M) = \mathrm{tf}(t,m) \cdot \mathrm{idf}(t,M)$$

Finally, the TRU for method m, $\mathrm{TRU}(m,M)$, is evaluated by calculating the harmonic mean of the TF-IDF values of all its tokens:

$$\mathrm{TRU}(m,M) = \frac{f_{t,m}}{\sum_{t' \in m} \frac{1}{\mathrm{tfidf}(t,m,M)}}$$

As before, $f_{t,m}$ is the raw count of token, t, in the method's representation. $\sum_{t' \in m} \frac{1}{\mathrm{tfidf}(t,m,M)}$ is the sum of the inverse TF-IDF values for all tokens in m.

We exclusively developed the TRU metric's conceptualisation, software vulnerability prediction contextualisation, hypothesis, design, and implementation, drawing inspiration from how the TF-IDF technique tends to assign higher weightings to rare words in information retrieval.

4.3.2 Token-Based Hit-Dependent Metrics

Table 4.4: Token-Based Hit-Dependent Metrics

Token-Based Hit-Dependent (Security-Relevant) Metric	**Abbr.**
Number of Hit Shingles	NHS
Number of Distinct Hit Shingles	NDHS
Number of Vulnerability Dataset Tokens	NVT
Number of Distinct Vulnerability Dataset Tokens	NDVT
Target Software System Method-to- ↪ Vulnerable Dataset Method Shingle Similarity Ratio	TVSSR
Shingle Hits-to-Target Software System Method Similarity Ratio	SHTSR
Shingle Hits-to-Vulnerable Dataset Method Similarity Ratio	SHVSR
Number of Shingle Matches	NUSM
Shingle Match Ratio	SMR

Table 4.5: Token-Based Hit Dependent Metrics Code Attributes of Concern

Token-Based Hit-Dependent Metric Abbr.	**Attributes of Concern**		
	Intricacy	*Similarity*	*Size*
NHS		✓	✓
NDHS	✓	✓	✓
NVT		✓	✓
NDVT	✓	✓	✓
TVSSR		✓	
SHTSR		✓	
SHVSR		✓	
NUSM		✓	
SMR		✓	

Many studies have utilised traditional software metrics, such as McCabe's Cyclomatic Complexity, Number of Lines of Code, Code Churn, and Fan-in and Fan-out dependency metrics, as features in vulnerability prediction models. However, these metrics are often criticised for not adequately capturing code semantics because they are not designed with security in mind. Their primary focus is on quantifying code characteristics, which many scholars believe contributes to their relatively poor performance in vulnerability prediction. Recent research emphasises the importance of using security-specific metrics for more effective vulnerability prediction [Shin and Williams, 2008b,a, Morrison et al., 2015, Munaiah et al., 2017, Sultana and Chong, 2019, Al Debeyan et al., 2022, Zimmermann et al., 2010, Shin et al., 2010, Doyle and Walden, 2011, Shin and Williams, 2013, Moshtari et al., 2013, Meneely et al., 2013, Walden et al., 2014, Perl et al., 2015, Younis et al., 2016, Sultana, 2017b, Sultana et al., 2018b, Chong et al., 2019b, Kalouptsoglou et al., 2020].

We aimed to develop security-aware metrics by leveraging the *knowledge* of known vulnerability patterns, creating nine additional (hit-dependent) metrics as presented in Table 4.4. Table 4.5 details the code attributes each metric captures. All hit-dependent metrics except NUSM and SMR measure the similarity between a target software system method and a known vulnerable method. For example, the NHS metric measures the similarity and size between a target software system method and a known vulnerable method, while the NDHS metric captures the diversity of code elements within this similarity. NUSM and SMR, on the other hand, measure the general distribution of similarities between a target software system method and known vulnerable methods.

A hit refers to the intersection of shingles between a target and vulnerable methods in a dataset. Unlike the hit-independent metrics, these nine hit-dependent metrics are pattern-based. Their evaluation depends on the patterns found in the shingles of a target software system method and a known vulnerable method, as facilitated by the hits.

Because vulnerable and non-vulnerable code elements often have similar attributes, which complicates their differentiation [Pereira et al., 2021], these hit-dependent metrics facilitate this differentiation by focusing on the patterns found in vulnerable code, making them security-relevant.

These security-relevant metrics aim to quantify the similarity of code components in the target system to those in a dataset of known vulnerabilities. These metrics are termed 'security-relevant' because they encode patterns found in vulnerable code rather than directly indicating the presence of vulnerabilities.

4.3.2.1 Number of Hit Shingles (NHS)

A target software system method's NHS is the number of shingles shared between it and the most similar known vulnerable method in a vulnerability dataset. This metric is straightforward, simply counting the shared shingles.

The hypothesis is that the more code representation elements a target software system method shares with a known vulnerable method, the more likely it is to exhibit the same vulnerability.

For a target software system method m_t with a *multiset* of shingles $S_t(m_t)$ and a matching method m_v from a vulnerability dataset with a *multiset* of shingles $S_v(m_v)$, NHS is expressed as:

$$\text{NHS} = |h|$$

Here, h represents the hits between $S_t(m_t)$ and $S_v(m_v)$, where $h = S_t(m_t) \cap S_v(m_v)$.

The hits are shingles instead of tokens because shingles encode the structure of the source code, capturing the context more effectively.

We exclusively conceptualised the NHS metric, contextualised it within software vulnerability prediction and developed its hypothesis, design and implementation.

4.3.2.2 Number of Distinct Hit Shingles (NDHS)

A target software system method's NDHS is the count of *unique* shingles shared between the method and the most similar known vulnerable method in a vulnerability dataset. NDHS is similar to NHS, but it measures the diversity of the shared shingles rather than their total number.

For a target software system method m_t with a *set* of shingles $S'_t(m_t)$ and a matching method m_v from a vulnerability dataset with a *set* of shingles $S'_v(m_v)$, let h' represent the hits between $S'_t(m_t)$ and $S'_v(m_v)$, where $h' = S'_t(m_t) \cap S'_v(m_v)$. NDHS is expressed as:

$$\text{NDHS} = |h'|$$

We exclusively conceptualised the NDHS metric, contextualised it within software vulnerability prediction and developed its hypothesis, design and implementation.

4.3.2.3 Number of Vulnerability Dataset Tokens (NVT)

A target software system method's NVT is the count of tokens in its most similar known vulnerable method from a vulnerability dataset. The NVT metric is similar to the NTT metric introduced in Subsubsection 4.3.1.1, which measures the number of tokens in a target software system method. However, while NTT measures the tokens of the target software system method, NVT measures the tokens of the most similar known vulnerable method. NTT is hit-independent, whereas NVT is hit-dependent because it requires a match with a known vulnerable method.

The hypothesis is that a method matching with a large and complex known vulnerable method is likely to be complex and potentially vulnerable.

NVT is expressed as:

$$\text{NVT} = |T_v(m_v)|$$

We exclusively conceptualised the NVT metric, contextualised it within software vulnerability prediction and developed its hypothesis, design and implementation.

4.3.2.4 Number of Distinct Vulnerability Dataset Tokens (NDVT)

A target software system method's NDVT is the count of *unique* tokens in its most similar known vulnerable method from a vulnerability dataset. The NDVT metric is similar to the NDTT metric. While NDTT measures the number of distinct tokens in a target software system method, NDVT measures the number of distinct tokens in the most similar known vulnerable method. NDTT is hit-independent, whereas NDVT is hit-dependent because its calculation requires a match with a known vulnerable method.

NDVT is expressed as:

$$\text{NDVT} = |T'_v(m_v)|$$

The NDVT metric was inspired by the NDTT metric, which in turn was inspired by several studies, including Shin et al. [2010], Giger et al. [2012], Morrison et al. [2015], Pascarella et al. [2018], and Du et al. [2019]. However, we conceptualised its information retrieval-based design and implementation.

4.3.2.5 Target Software System Method-to-Vulnerable Dataset Method Shingle Similarity Ratio (TVSSR)

A target software system method's TVSSR is the Jaccard Similarity of its shingles to those of the most similar known vulnerable method in a vulnerability dataset.

The TVSSR metric measures the extent to which a target software system method shares code elements with a known vulnerable method. It evaluates the distinct shingles appearing in both methods relative to the total number of distinct shingles between them. While the metric does not directly consider order and frequency, the shingles inherently encode these aspects.

Jaccard Similarity is the ratio of the intersection to the union of two sets. In this context, it refers to the ratio of shared distinct shingles between m_t and m_v to the total distinct shingles in both methods.

The hypothesis is that the more unique code representation elements a method shares with a known vulnerable method, the more likely it is to exhibit the same vulnerability.

For a target software system method m_t with a *set* of shingles $S'_t(m_t)$ and a matching method m_v from a vulnerability dataset with a *set* of shingles $S'_v(m_v)$, let h' represent the hits between $S'_t(m_t)$ and $S'_v(m_v)$, where $h' = S'_t(m_t) \cap S'_v(m_v)$. TVSSR is expressed as:

$$\text{TVSSR} = \frac{|h'|}{|S'_t(m_t) \cup S'_v(m_v)|}$$

Or, more derivatively:

$$\text{TVSSR} = \frac{NDHS}{(NDTS+NDVS)-NDHS}$$

$NDTS$ and $NDVS$ are temporary variables representing the number of distinct shingles in the target and known vulnerable methods, respectively.

We exclusively developed the TVSSR metric's conceptualisation, software vulnerability prediction contextualisation, hypothesis, design, and implementation, drawing inspiration from the Jaccard Similarity technique used in string metrics.

4.3.2.6 Shingle Hits-to-Target Software System Method Similarity Ratio (SHTSR)

The SHTSR of a target software system method is the ratio of shared shingles between its shingles and those of the most similar known vulnerable method to the total number of shingles in the target software system method. This metric measures the extent to which a target software system method comprises shingles shared with a known vulnerable method.

The hypothesis is that the more a target software system method includes code representation elements present in a known vulnerable method, the more likely it is to exhibit the same vulnerability.

For a target software system method m_t with a *set* of shingles $S'_t(m_t)$ and a matching method m_v from a vulnerability dataset with a *set* of shingles $S'_v(m_v)$, let h' represent the hits between $S'_t(m_t)$ and $S'_v(m_v)$, where $h' = S'_t(m_t) \cap S'_v(m_v)$. SHTSR is expressed as:

$$\text{SHTSR} = \frac{|h'|}{|S'_t(m_t)|}$$

Or, more derivatively:

$$\text{SHTSR} = \frac{NDHS}{NDTS}$$

NDTS is a temporary variable representing the number of distinct shingles in the target software system method.

We exclusively conceptualised the SHTSR metric, contextualised it within software vulnerability prediction and developed its hypothesis, design and implementation.

4.3.2.7 Shingle Hits-to-Vulnerable Dataset Method Similarity Ratio (SHVSR)

A target software system method's SHVSR is the ratio of the shared shingles between its shingles and those of the most similar known vulnerable method to the total number of shingles in the vulnerable method. This metric measures how much a known vulnerable method comprises shingles shared with a target software system method.

SHVSR is similar to SHTSR, with the difference being that SHVSR evaluates the vulnerable method, while SHTSR evaluates the target software system method.

When evaluating SHTSR, the question is: "To what extent do the unique shingles in our hits constitute the unique shingles in our target software system method?" For SHVSR, the question is: "To what extent do the unique shingles in our hits constitute the unique shingles in the vulnerable method?"

The hypothesis is that the more a vulnerable method comprises code representation elements present in a target software system method, the more likely it is that the target software system method's components will exhibit the same vulnerability.

For a target software system method m_t with a *set* of shingles $S'_t(m_t)$, and a matching method m_v from a vulnerability dataset with a *set* of shingles $S'_v(m_v)$, let h' represent the hits between $S'_t(m_t)$ and $S'_v(m_v)$, where $h' = S'_t(m_t) \cap S'_v(m_v)$. SHVSR is expressed as:

$$\text{SHVSR} = \frac{|h'|}{|S'_v(m_v)|}$$

Or, more derivatively:

$$\text{SHVSR} = \frac{NDHS}{NDVS}$$

NDVS is a temporary variable representing the number of distinct shingles in the known vulnerable method.

We exclusively conceptualised the SHVSR metric, contextualised it within software vulnerability prediction and developed its hypothesis, design and implementation.

4.3.2.8 Number of Shingle Matches (NUSM)

A target software system method's NUSM is the count of known vulnerable methods in a vulnerability dataset that share at least one shingle with the method in question. This metric measures the number of known vulnerable methods that share shingles with a target software system method.

The hypothesis is that the more vulnerable methods that share at least one code representation element with a target software system method, the more likely the target software system method is to be vulnerable.

For a target software system method m_t with a *set* of shingles $S'_t(m_t)$ and a vulnerability dataset V containing n known vulnerable methods $m_{v_1}, m_{v_2}, \ldots, m_{v_n}$, each with *sets* of shingles $S'_{v_1}(m_{v_1}), S'_{v_2}(m_{v_2}), \ldots, S'_{v_n}(m_{v_n})$, NUSM is expressed as:

$$\text{NUSM} = |\{m_{v_i} \in V \,:\, S'_t(m_t) \cap S'_{v_i}(m_{v_i}) \neq \emptyset\}|$$

We exclusively conceptualised the NUSM metric, contextualised it within software vulnerability prediction and developed its hypothesis, design and implementation.

4.3.2.9 Shingle Match Ratio (SMR)

A target software system method's SMR is the ratio of known vulnerable methods in a vulnerability dataset that share at least one shingle with the method to the total number of vulnerable methods in the dataset. The SMR metric is similar to the NUSM metric but focuses on the ratio rather than the absolute number. In other words, a method's SMR is its NUSM divided by the total number of vulnerable methods in the dataset.

For a target software system method m_t with a *set* of shingles $S'_t(m_t)$ and a vulnerability dataset V containing n known vulnerable methods $m_{v_1}, m_{v_2}, \ldots, m_{v_n}$, each with *sets* of shingles $S'_{v_1}(m_{v_1}), S'_{v_2}(m_{v_2}), \ldots, S'_{v_n}(m_{v_n})$, SMR is expressed as:

$$\text{SMR} = \frac{|\{m_{v_i} \in V : S'_t(m_t) \cap S'_{v_i}(m_{v_i}) \neq \emptyset\}|}{n}$$

Or, more derivatively:

$$\text{SMR} = \frac{NUSM}{n}$$

We exclusively conceptualised the SMR metric, contextualised it within software vulnerability prediction and developed its hypothesis, design and implementation.

The following subsection illustrates how these metrics are calculated using a hypothetical target software system and a hypothetical vulnerable method.

4.3.3 Token-Based Metrics Calculation: An Illustration

We illustrate the calculation of the metrics developed in this study using one of the most well-known software vulnerabilities: SQL Injection[1] [2].

SQL Injection is a code injection technique exploiting vulnerabilities in an application's software layer. It occurs when user input is not correctly filtered for string literal escape characters embedded in SQL statements, allowing an attacker to execute arbitrary SQL commands. This vulnerability is common and poses a significant risk to web applications, potentially leading to data breaches, loss, and other security incidents.

We use two code snippets to illustrate the calculation of the token-based metrics: a hypothetical target software system method and a hypothetical known vulnerable method from a vulnerability dataset.

For simplicity, we exclude specific metrics requiring more extensive information than what is representable within a method body. These metrics include TICC, TRICC, NTDT, NTDDT, TRU, NUSM, and SMR. For example, TICC, TRICC, NTDT, and NTDDT require information on an entire method's change history,

[1] `https://owasp.org/www-community/attacks/SQL_Injection`
[2] `https://cwe.mitre.org/data/definitions/89.html`

which is neither representable within a method body nor in its token representation and shingles.

The target software system method in Listing 4.2 is a simple Java method that prompts the user to enter a username, constructs a SQL query using the username, and processes the query. This method is vulnerable to SQL Injection. It constructs an SQL query using user input without proper filtering, allowing an attacker to enter a malicious username containing SQL commands.

Similarly, the vulnerability dataset method in Listing 4.3 is another simple Java method that constructs a SQL query using a username parameter and then executes the query. Like the target software system method, it is vulnerable to SQL Injection due to the lack of proper input filtering, allowing an attacker to execute malicious SQL commands.

Listing 4.2: `processUsername` Method

```java
        public void processUsername() throws SQLException {
            Scanner scanner = new Scanner(System.in);
            System.out.println("Enter your username:");
            String username = scanner.nextLine();
            String q = "SELECT * FROM users WHERE username = '" + username + "'";
            processQuery(q);
            scanner.close();
        }
```

```
        public void processUsername ( ) throws SQLException {
            ↪ Scanner scanner = new Scanner ( System . in ) ;
            ↪ System . out . println ( " Enter your username : "
            ↪ ) ; String username = scanner . nextLine ( ) ;
            ↪ String q = " SELECT * FROM users WHERE username = '
            ↪  " + username + " ' " ; processQuery ( q ) ;
            ↪ scanner . close ( ) ; }
```

Figure 4.4: Single Whitespace-Separated Token Representation of the Method in Listing 4.2

Figure 4.4 shows the single whitespace-separated token representation of the `processUsername` method in Listing 4.2. Some vulnerabilities can occur within a string literal, such as the SQL Injection vulnerability in the target software system method or even within a single token, such as a hardcoded password or API key. For this reason, we aimed for ultimate granularity in our token representation by disintegrating code elements, including strings, into simple individual tokens to optimise token retrieval (i.e., using information retrieval) and analysis.

Table 4.6: Shingles of the Method in Listing 4.2

`public void processUsername ( )`	`= scanner . nextLine (`
`void processUsername ( ) throws`	`scanner . nextLine ( )`
`processUsername ( ) throws SQLException`	`. nextLine ( ) ;`
`( ) throws SQLException {`	`nextLine ( ) ; String`
`) throws SQLException { Scanner`	`( ) ; String q`
`throws SQLException { Scanner scanner`	`) ; String q =`
`SQLException { Scanner scanner =`	`; String q = "`
`{ Scanner scanner = new`	`String q = " SELECT`
`Scanner scanner = new Scanner`	`q = " SELECT *`
`scanner = new Scanner (`	**`= " SELECT * FROM`**
`= new Scanner ( System`	**`" SELECT * FROM users`**
`new Scanner ( System .`	**`"SELECT * FROM users WHERE`**
`Scanner ( System . in`	**`* FROM users WHERE username`**
`( System . in )`	**`FROM users WHERE username =`**
`System . in ) ;`	**`users WHERE username = '`**
`. in ) ; System`	**`WHERE username = ' "`**
`in ) ; System .`	**`username = ' " +`**
`) ; System . out`	**`= ' " + username`**
`; System . out .`	**`' " + username +`**
`System . out . println`	**`" + username + "`**
`. out . println (`	**`+ username + " '`**
`out . println ( "`	**`username + " ' "`**
`. println ( " Enter`	**`+ " ' " ;`**
`println ( " Enter your`	`" ' " ; processQuery`
`( " Enter your username`	`' " ; processQuery (`
`" Enter your username :`	`" ; processQuery ( q`
`Enter your username : "`	`; processQuery ( q )`
`your username : " )`	`processQuery ( q ) ;`
`username : " ) ;`	`( q ) ; scanner`
`: " ) ; String`	`q ) ; scanner .`
`" ) ; String username`	`) ; scanner . close`
`) ; String username =`	`; scanner . close (`
`; String username = scanner`	`scanner . close ( )`
`String username = scanner .`	`. close ( ) ;`
`username = scanner . nextLine`	`close ( ) ; }`

Table 4.6 shows the shingles (5-grams) derived from the token representation of the target software system method in Figure 4.4.

Shingles typically exclude punctuation and whitespace characters in natural language processing, focusing only on words. However, in our source code context, we include all code elements (except comments and whitespaces), such as brackets, parentheses, and semicolons, to capture the code's structure and syntax.

Listing 4.3: `getUserData` Method

```java
public ResultSet getUserData(String username) {

    String dbUrl = getDatabaseUrl();
    ResultSet resultSet = null;

    try (Connection conn = DriverManager.getConnection(dbUrl);
        Statement statement = conn.createStatement()) {

        String query = "SELECT * FROM users WHERE username = '" + username + "'";
        resultSet = statement.executeQuery(query);

    } catch (SQLException e) {
        e.printStackTrace();
    }
    return resultSet;
}
```

```
public ResultSet getUserData ( String username ) { String
↪  dbUrl = getDatabaseUrl ( ) ; ResultSet resultSet =
↪  null ; try ( Connection conn = DriverManager .
↪ getConnection ( dbUrl ) ; Statement statement =
↪ conn . createStatement ( ) ) { String query = "
↪ SELECT * FROM users WHERE username = ' " + username
↪  + " ' " ; resultSet = statement . executeQuery (
↪ query ) ; } catch ( SQLException e ) { e .
↪ printStackTrace ( ) ; } return resultSet ; }
```

Figure 4.5: Single Whitespace-Separated Token Representation of the Method in Listing 4.3

Table 4.7: Shingles of the Method in Listing 4.3

`public ResultSet getUserData ( String`	`query = " SELECT *`
`ResultSet getUserData ( String username`	**`= " SELECT * FROM`**
`getUserData ( String username )`	**`" SELECT * FROM users`**
`( String username ) {`	**`"SELECT * FROM users WHERE`**
`String username ) { String`	**`* FROM users WHERE username`**
`username ) { String dbUrl`	**`FROM users WHERE username =`**
`) { String dbUrl =`	**`users WHERE username = '`**
`{ String dbUrl = getDatabaseUrl`	**`WHERE username = ' "`**
`String dbUrl = getDatabaseUrl (`	**`username = ' " +`**
`dbUrl = getDatabaseUrl ( )`	**`= ' " + username`**
`= getDatabaseUrl ( ) ;`	**`' " + username +`**
`getDatabaseUrl ( ) ; ResultSet`	**`" + username + "`**
`( ) ; ResultSet resultSet`	**`+ username + " '`**
`) ; ResultSet resultSet =`	**`username + " ' "`**
`; ResultSet resultSet = null`	**`+ " ' " ;`**
`ResultSet resultSet = null ;`	`" ' " ; resultSet`
`resultSet = null ; try`	`' " ; resultSet =`
`= null ; try (`	`" ; resultSet = statement`
`null ; try ( Connection`	`; resultSet = statement .`
`; try ( Connection conn`	`resultSet = statement . executeQuery`
`try ( Connection conn =`	`= statement . executeQuery (`
`( Connection conn = DriverManager`	`statement . executeQuery ( query`
`Connection conn = DriverManager .`	`. executeQuery ( query )`
`conn = DriverManager . getConnection`	`executeQuery ( query ) ;`
`= DriverManager . getConnection (`	`( query ) ; }`
`DriverManager . getConnection ( dbUrl`	`query ) ; } catch`
`. getConnection ( dbUrl )`	`) ; } catch (`
`getConnection ( dbUrl ) ;`	`; } catch ( SQLException`
`( dbUrl ) ; Statement`	`} catch ( SQLException e`
`dbUrl ) ; Statement statement`	`catch ( SQLException e )`
`) ; Statement statement =`	`( SQLException e ) {`
`; Statement statement = conn`	`SQLException e ) { e`
`Statement statement = conn .`	`e ) { e .`
`statement = conn . createStatement`	`) { e . printStackTrace`
`= conn . createStatement (`	`{ e . printStackTrace (`
`conn . createStatement ( )`	`e . printStackTrace ( )`
`. createStatement ( ) )`	`. printStackTrace ( ) ;`
`createStatement ( ) ) {`	`printStackTrace ( ) ; }`
`( ) ) { String`	`( ) ; } return`
`) ) { String query`	`) ; } return resultSet`
`) { String query =`	`; } return resultSet ;`
`{ String query = "`	`} return resultSet ; }`
`String query = " SELECT`	

Table 4.7 shows the shingles (5-grams) derived from the token representation in Figure 4.5 of the `getUserData` method in Listing 4.3. Again, the shingles include every code element (except comments and whitespaces), such as brackets, parentheses, and semicolons, to capture the code's structure and every low-level syntactic detail.

Table 4.8: Example Token-Based Metrics Calculation

Hit-Independent Metrics	**Metric Value**
NTT	74
NDTT	35
Hit-Dependent Metrics	**Metric Value**
NHS	14
NDHS	14
NVT	89
NDVT	37
TVSSR	$\frac{14}{141}$
SHTSR	$\frac{14}{70}$
SHVSR	$\frac{14}{85}$

Table 4.8 presents the calculated values of the metrics for the `processUsername` Method and the hypothetical vulnerable method, as derived from a vulnerability dataset. The full names of the metrics are in Table 4.1 and Table 4.4.

The NTT metric, the number of tokens in the target software system method, is 74, obtained by counting the tokens in the token representation of the target software system method (Figure 4.4).

The NDTT metric, the number of *distinct* tokens in the target software system method, is 35, obtained by counting the distinct tokens in the token representation of the target software system method (Figure 4.4).

The NHS metric, the number of shared shingles between the target software system method and the vulnerable method, is 14, determined by counting the shingles that appear in both methods (Tables 4.6 and 4.7). For clarity, these shared shingles are emboldened in both tables.

The NDHS metric, the number of *distinct* shared shingles between the target and vulnerable methods, is also 14.

The NVT metric, the number of tokens in the vulnerable method, is 89, obtained by counting the tokens in the token representation of the vulnerable method (Figure 4.5).

The NDVT metric, the number of *distinct* tokens in the vulnerable method, is 37, obtained by counting the distinct tokens in the token representation of the vulnerable method (Figure 4.5).

The TVSSR metric, the Jaccard Similarity of the target software system method's shingles and the vulnerable method's shingles is $\frac{14}{(70+85)-14}$ or $\frac{14}{141}$, where 70 and 85 are the number of *distinct* shingles in the target and vulnerable methods, respectively (Tables 4.6 and 4.7).

The SHTSR metric, which is the ratio of the number of *distinct* shared shingles to the number of *distinct* shingles in the target software system method, is $\frac{14}{70}$.

The SHVSR metric, which is the ratio of the number of *distinct* shared shingles to the number of *distinct* shingles in the vulnerable method, is $\frac{14}{85}$.

This example illustrates how we calculate the metrics developed in this study using a hypothetical target software system and a hypothetical vulnerable method. The following section details our methodology, including how these metrics are leveraged in our machine learning classification.

4.4 Methodology

This section presents the chapter's methodology. We provide preliminary information on our approach, followed by detailed information on our datasets, data preprocessing, information retrieval techniques, and machine learning analysis.

4.4.1 Overview of the Methodology

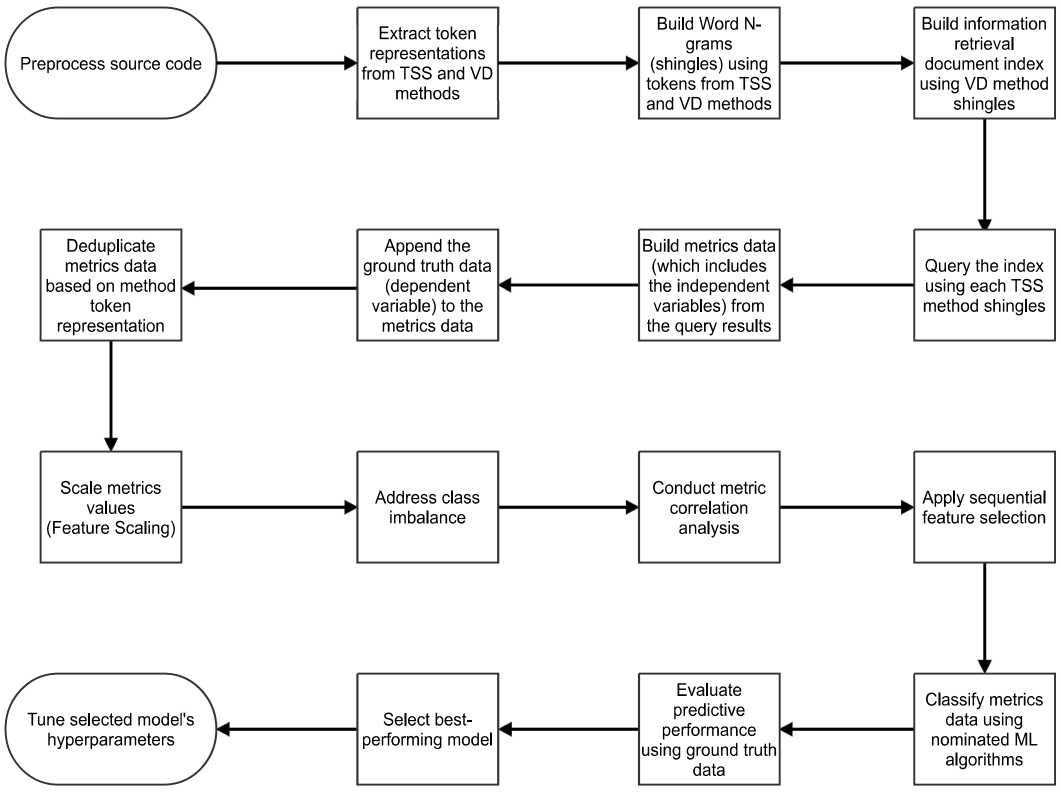

Preprocess source code
Extract token representations from TSS and VD methods
Build Word N-grams (shingles) using tokens from TSS and VD methods
Build information retrieval document index using VD method shingles
Query the index using each TSS method shingles
Build metrics data (which includes the independent variables) from the query results
Append the ground truth data (dependent variable) to the metrics data
Deduplicate metrics data based on method token representation
Scale metrics values (Feature Scaling)
Address class imbalance
Conduct metric correlation analysis
Apply sequential feature selection
Classify metrics data using nominated ML algorithms
Evaluate predictive performance using ground truth data
Select best-performing model
Tune selected model's hyperparameters

Figure 4.6: Token-Based Vulnerability Prediction Methodology Overview (Within-Project)

Figure 4.6 provides a high-level summary of our approach, grouped into six main phases: source code preprocessing, token representation extraction, word n-grams (shingles) generation, information retrieval, metrics development, and machine learning analysis.

4.4.1.1 Source Code Preprocessing

Following the acquisition of the dataset, we began by preprocessing the source code of our target software system and the vulnerability dataset. The target software system is the program in which we want to predict vulnerabilities.

This phase involved several preparatory steps, including identifying and filtering out irrelevant code artefacts, such as abstract and test methods. We also removed source code comments from all files to ensure they did not influence the analysis.

4.4.1.2 Source Code Token Representation Extraction

After preprocessing, we parsed each source code file in our target software system and the vulnerability dataset. We used JavaParser[3] to extract the token representations of methods from both software systems. JavaParser is a Java library that parses Java source code and generates ASTs from the parsed code.

4.4.1.3 Word N-grams (Shingles) Generation

We generated shingles for each method using the token representations of methods in our target software system and the vulnerability dataset with Apache Lucene's ShingleFilter[4]. These shingles were then utilised in the next phase of our experiment, which involved information retrieval. Apache Lucene is a Java-based, high-performance, full-featured text search engine library, which we will discuss in more detail later in this section.

4.4.1.4 Information Retrieval

This phase involved constructing a document index and querying it. The aim was to apply information retrieval techniques to the shingles generated from the extracted source code token representations of the methods in our target software system and the vulnerability dataset.

1. *Document Index Construction*: We used the shingles generated from the vulnerability dataset methods to build an information retrieval document index. This index serves as a data repository, facilitating the efficient storage and retrieval of relevant information, where a 'document' refers to the shingles generated from a method's token representation.

[3]`https://javaparser.org/`
[4]`https://lucene.apache.org/`

2. *Document Index Querying*: We used the shingles from each target software system method to query and retrieve the most similar methods from the document index.

4.4.1.5 Metrics Data Development

This experiment phase embodies our core contribution: the sixteen metrics we developed using our novel information retrieval-driven approach to vulnerability prediction. This phase had two subphases:

1. *Metrics Calculation and Feature Engineering*: In this subphase, we generated the metrics data for each method in our target software system. This involved leveraging the attributes from the source code token representations and shingles of the methods, as well as the results from the information retrieval phase. The metrics data included two categories: hit-independent (seven metrics) and hit-dependent (nine metrics). The hit-independent metrics were calculated using only the attributes of the target software system methods. In contrast, the hit-dependent metrics used attributes from the target software system methods and their most similar methods in the vulnerability dataset. Hit-independent metrics communicated the methods' structural and evolutionary details, while hit-dependent metrics provided security-aware data by leveraging the vulnerability dataset. This process constituted the feature engineering for our machine learning classification.

2. *Ground Truth Data Appendation*: This subphase involved supplementing the metrics data with ground truth information for each method in our target software system. The ground truth data, comprising vulnerability fix information from the official security reports, was crucial for evaluating the performance of our machine learning classification.

4.4.1.6 Machine Learning Analysis

This phase aimed to:

i identify the best-performing machine learning classifier for vulnerability prediction;

ii identify the best-performing combination of metrics for vulnerability prediction.

We completed several subphases to achieve these aims:

1. *Metrics Data Deduplication*: This subphase aimed to eliminate data leakage, which can cause overfitting and inflated performance metrics.

2. We removed duplicate data by ensuring a unique token representation per method per release.

3. *Feature Scaling*: We normalised the metrics data using the Min-Max feature scaling technique to ensure all metrics values were on the same scale and contributed equally to the machine learning classification.

4. *Class Imbalance Mitigation*: Due to the class imbalance between vulnerable and non-vulnerable code artefacts, we applied the Synthetic Minority Oversampling Technique (SMOTE) to balance the metrics data before training.

5. *Correlation Analysis*: This subphase identified correlations among the metrics (independent variables) and between the metrics and the ground truth (dependent variable).

6. *Feature Selection*: We applied the Sequential Feature Selection (SFS) feature selection technique to identify the best-performing combination of metrics for vulnerability prediction.

7. *Classification*: We trained ten machine learning classifiers using the metrics data as independent variables, and the vulnerability fix information as the dependent variable. The goal was to predict whether each method in our target software system was vulnerable or non-vulnerable.

8. *Performance Evaluation*: We evaluated the classifiers' performance using the ground truth data, calculating precision, recall, and F1 score.

9. *Model Selection*: We selected the best-performing classifier based on the highest F1 score.

10. *Model Optimisation (Hyperparameter Tuning)*: Using the Grid Search hyperparameter tuning technique, we tuned the hyperparameters of the best-performing classifier to improve its performance.

The above enumeration summarises the methodology used in this study. The following subsections provide detailed information on each overviewed item.

4.4.2 Dataset

Our analysis involved a target software system and vulnerability datasets. The target software system dataset was the focus of our vulnerability prediction analysis. In contrast, the vulnerability dataset was crucial for assessing the likelihood of vulnerability of the methods in the target system.

Our target software system was Apache Tomcat 7[5], acquired in the fourth quarter of 2021. The vulnerability dataset was the National Institute of Standards and Technology (NIST) Software Assurance Reference Dataset (SARD)[6], obtained in the fourth quarter of 2022. We used SARD to evaluate whether methods in various Apache Tomcat 7 releases exhibited the same vulnerable patterns as those in the vulnerability dataset.

4.4.2.1 Target Software System

We used Apache Tomcat 7 as our target software system. Tomcat is an open-source implementation of Java Servlet, JavaServer Pages, Java Expression Language, and WebSocket technologies. It provides a fully Java-based HTTP web server environment for running Java code. The project is developed by an open-source community with support from the Apache Software Foundation and is licensed under Apache License 2.0.

We chose this software system for several reasons, including access to its complete source code, security reports, and information on vulnerability fixes. These resources, available on the Apache Tomcat website, are essential for evaluating predictions.

[5] `https://archive.apache.org/dist/tomcat/tomcat-7/`
[6] `https://samate.nist.gov/SARD/`

We analysed several releases of Apache Tomcat 7, between 7.0.0 and 7.0.108. We excluded a few source code files that we could not parse and test or other non-conventional methods due to their irrelevance in our context.

4.4.2.2 Ground Truth Data

We used Apache Tomcat 7 fixed vulnerabilities data[7], spanning releases 7.0.2 to 7.0.108, as our ground truth.

We filtered out two groups of components from our ground truth data:

1. Test-related methods that are modified as part of a vulnerability fix since they do not directly define a system's functionality or contribute to its vulnerability proneness.

2. Methods added from scratch as part of a vulnerability fix, as they did not exist before the fix and could not have contributed to pre-existing vulnerabilities. While such methods could introduce new vulnerabilities, our analysis did not consider this risk category. We focused solely on the threats posed by partially fixed, pre-existing methods.

4.4.2.3 National Institute of Standards and Technology (NIST) Software Assurance Reference Dataset

The SARD vulnerability dataset provided numerous vulnerable code samples. It contained over 450,000 test programs with documented weaknesses, ranging from small synthetic to large applications at the time of acquisition. While we focused on Java-based programs, the dataset includes C, C++, Java, PHP, and C#, covering over 150 Common Weakness Enumeration (CWE)[8] classes.

A typical SARD test case comprises one or more source code files with attributes such as 'type', 'author', 'language', 'state', 'status', and 'submission date.' In our experiment, the 'type' was 'source code', and the 'language' was 'Java.'

[7]`https://tomcat.apache.org/security-7.html`
[8]`https://cwe.mitre.org/`

The ‘state’ attribute options are ‘good’, ‘bad’, and ‘mixed.’ A ‘good’ test case is non-vulnerable (false positive), a ‘bad’ test case is vulnerable (true positive), and a ‘mixed’ test case contains both vulnerable and non-vulnerable code.

The ‘status’ attribute options are ‘candidate’, ‘accepted’, and ‘deprecated.’ Test cases start as ‘candidate’ and, if they pass a review by a SARD librarian, become ‘accepted.’ Accepted test cases meet quality standards, are well-documented, represent specific weaknesses, are easy to understand, and are free of ambiguities.

We prioritised ‘accepted’ test cases due to their quality and credibility. However, due to data scarcity, availability and accessibility-related challenges in software vulnerability research, we also included ‘candidate’ test cases to increase the number of vulnerable samples. We excluded ‘deprecated’ test cases as SARD discourages their use.

Also, we only used test cases from the ‘bad’ and ‘mixed’ categories for the ‘state’ attribute. We extracted the vulnerable methods from the ‘mixed’ category and skipped the non-vulnerable counterparts.

Table 4.9: Token-Based Single Software System Dataset Details

Description	**Value**
Total Number of Analysed Target Software System Releases	76
Total Number of Analysed Vulnerability Fixes	261
Total Number of Indexed Vulnerability Dataset Methods	20,692

Table 4.9 presents the details of our target software system and vulnerability dataset. We analysed 76 releases of Apache Tomcat 7 and 261 vulnerability fixes. The SARD vulnerability dataset comprised 20,692 methods.

4.4.3 Data Preprocessing

We parsed and extracted source code tokens using JavaParser and generated shingles from these tokens using Apache Lucene’s ShingleFilter. Using information retrieval techniques, we developed features (metrics) for machine learning classification from the extracted tokens and generated shingles.

4.4.3.1 Source Code Token Extraction

Source code tokenisation involves converting source code into a sequence of tokens that represent its structure. We extracted tokens from all analysed methods in our target software system and the vulnerability dataset.

4.4.3.2 Shingle Generation

We generated shingles for each method in our target software system and the vulnerability dataset following token extraction.

Shingling breaks down code into smaller, overlapping fragments called shingles. This technique is employed in text analysis for detecting plagiarism and analysing code similarity.

The ideal shingle size depends on the context and nature of the analysis. Smaller shingles capture fine details but may produce noise and be computationally expensive. Larger shingle sizes reduce noise but may miss subtle details. We conducted preliminary experiments and found that a shingle size of '5' was ideal for our analysis.

After token extraction and shingle generation, we applied information retrieval techniques and machine learning classification, which were discussed later in this section. Before that, we describe additional steps in preparing our data: deduplication, ground truth evaluation, feature scaling and class imbalance mitigation.

4.4.3.3 Data Deduplication

We deduplicated the Tomcat (training) dataset to avoid data leakage[9] during machine learning classification. Data leakage occurs when a classifier inadvertently uses test data during training, which can happen if duplicates are not removed, making some data points appear in both the training and test sets.

Our deduplication strategy retained only the first release of each method in the dataset and all subsequent releases where its token representation changed. Figure 4.3 and Table 4.10 illustrate this strategy. Table 4.11 presents the post-

[9] `https://scikit-learn.org/stable/common_pitfalls.html#data-leakage`

deduplication figures of our target software system dataset, Apache Tomcat 7, showing a highly imbalanced class distribution.

Table 4.10: Deduplication Strategy

Method	**Release 1**	**Release 2**	**Release 3**	**Release 4**	**Release 5**
A	Include	Remove	Remove	Remove	Remove
B	Include	Include	Remove	Include	Remove
C	Include	Remove	Include	Remove	Remove
D	NA	Include	Remove	Remove	Remove
E	NA	Include	Include	Remove	NA

Table 4.11: Apache Tomcat 7 Post-Deduplication Dataset Details

Description	**Value**	**%**
Number of Non-Vulnerable Methods Post-Deduplication	25,166	98.52
Number of Vulnerable Methods Post-Deduplication	378	1.48
Total Number of Methods Post-Deduplication	25,544	

4.4.3.4 Ground Truth: Affected Methods Estimation

To measure the number of affected methods for each vulnerability fix, we assume that each fixed method reported in the ground truth was vulnerable in its earlier releases before the fix.

A vulnerability fix can affect multiple methods in a file and multiple files in a software system. The value 378 in Table 4.11 represents the number of affected methods, calculated from the 261 vulnerability fixes reported in Table 4.9.

4.4.3.5 Feature Scaling

Feature scaling is an essential preprocessing step in machine learning classification. It ensures that all features have a uniform scale, preventing model bias towards features with larger values.

We applied Min-Max scaling to standardise our metrics data. This technique transforms each feature into a fixed range of $[0,1]$ by subtracting the minimum value and dividing by the feature's range, as defined by:

$$X_{\text{scaled}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

Where X is the original value, X_{min} is the minimum value of the feature, and X_{max} is the maximum value of the feature.

This normalisation ensures that all feature values contribute equally to the learning process, improving model convergence and stability.

4.4.3.6 Software System Vulnerabilities and Data Imbalance

We addressed the class imbalance in our target software system dataset after deduplication.

Typically, a software system has far fewer vulnerable artefacts than non-vulnerable ones because the number of known vulnerabilities is usually low at any given time. This imbalance creates a skewed class distribution when comparing vulnerable artefacts to non-vulnerable ones [Ban et al., 2019, Shu et al., 2022, Liu et al., 2019]. In our case, the imbalance between vulnerable and non-vulnerable methods was extreme, as shown in Table 4.11. We addressed this imbalance before applying machine learning classification to ensure fairness and avoid model bias.

Chawla et al. [2002] introduced SMOTE to address the class imbalance problem by oversampling the minority class. SMOTE generates synthetic minority-class samples by interpolating between them, effectively balancing the class distribution.

We applied SMOTE to our dataset, oversampling the minority class (vulnerable methods) to balance the class distribution before proceeding to the machine learning classification phase.

4.4.4 Information Retrieval

Following source code token extraction and shingle generation, as described in Subsubsections 4.4.3.1 and 4.4.3.2, we applied information retrieval to the generated shingles to identify the best-matching methods in the vulnerability dataset for each method in the target software system. By ‘best-matching’, we mean shingle matches shared between the target software system method and the SARD methods shingles. To find these matches, we used Apache Lucene.

Apache Lucene is a Java-based open-source information retrieval library. It is a high-performance, full-featured text search engine that indexes and searches the entire text of a document. Lucene is mature, well-established, and highly scalable. It is capable of efficiently indexing and searching hundreds of millions of documents. It is highly configurable and extensible, supporting multiple platforms, including Windows, Linux, and macOS.

4.4.4.1 Apache Lucene Document Index Construction

Our Lucene index comprised 20,692 documents, each representing a method in the SARD dataset. The documents in the index were the shingles obtained from the token representation of the methods in the SARD dataset, as shown in Table 4.9. The index structure maps each method's custom ID to its shingles.

4.4.4.2 Apache Lucene Query Construction (BooleanQuery)

Our queries comprised shingles generated from the token representations of the target software system (Tomcat) methods. Each query included the method's shingles separated by the 'OR' Boolean logical operator.

Listing 4.4: `getChannelSendOptions` Method

```java
public int getChannelSendOptions() {
    return channelSendOptions;
}
```

```
public int getChannelSendOptions ( ) { return
    ↪ channelSendOptions ; }
```

Figure 4.7: Token Representation of the Method in Listing 4.4

```
public int getChannelSendOptions ( )↔int
    ↪ getChannelSendOptions ( ) {↔getChannelSendOptions
    ↪ ( ) { return↔( ) { return channelSendOptions↔) {
    ↪ return channelSendOptions ;↔{ return
    ↪ channelSendOptions ; }
```

Figure 4.8: Shingle Representation of the Method in Listing 4.4

For example, Listing 4.4 presents a method from our target software system. Figure 4.7 shows the token representation of the method. Figure 4.8 presents the shingles of the method, with a shingle size of five, and each shingle separated by ↔.

```
"public int getChannelSendOptions ( )" OR "int
    ↪ getChannelSendOptions ( ) {" OR "
    ↪ getChannelSendOptions ( ) { return" OR "( ) {
    ↪ return channelSendOptions" OR ") { return
    ↪ channelSendOptions ;"
```

Figure 4.9: Query String for the Method in Listing 4.4

Figure 4.9 shows the query string for the method in Listing 4.4. The 'OR' operator in our query strings makes each shingle optional, allowing Lucene to flexibly retrieve the best matching methods from the vulnerability dataset for any target software system method. The more ORs that match between a target software system method and a SARD method, the more relevant the SARD method is to the target software system method. We used the query results to calculate the hit-dependent metrics described in Subsection 4.3.2.

Table 4.12: Percentage of Vulnerable versus Non-Vulnerable Methods with Hits

Number of Vulnerable Methods	378
Number of Vulnerable Methods with Hits	361
% of Vulnerable Methods with Hits	95.50
Number of Non-Vulnerable Methods	25,166
Number of Non-Vulnerable Methods with Hits	15,201
% of Non-Vulnerable Methods with Hits	60.40

Table 4.12 extends Table 4.11, detailing the numbers and percentages of vulnerable and non-vulnerable methods with hits. The table shows that vulnerable methods are likelier to share patterns with known vulnerable methods in the dataset than non-vulnerable methods.

4.4.5 Machine Learning Analysis

After processing the token representations and shingles and calculating the metrics data, we fed the data into several machine learning classification algorithms (see Subsection 4.4.3). This part of the experiment utilised metrics derived from tokens and shingles to classify the methods in our target software system as either vulnerable or non-vulnerable.

After classification, we evaluated the performance of each algorithm using the ground truth data (see Subsubsections 4.4.2.2 and 4.4.3.4).

4.4.5.1 Nominated Classification Algorithms

We selected ten classifiers for our experiment based on their applicability, suitability for our dataset, and prevalence in the literature to facilitate comparability between studies. The classifiers include:

- AdaBoost classifier[10]
- Decision Tree classifier[11]
- Gaussian Naïve Bayes[12]
- Gradient Boosting classifier[13]
- K-Nearest Neighbors classifier[14]
- LightGBM classifier[15]
- Linear Support Vector classifier[16]
- Logistic Regression[17]
- Random Forest classifier[18]

[10]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

[11]https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

[12]https://scikit-learn.org/stable/modules/naive_bayes.html#gaussian-naive-bayes

[13]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html

[14]https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

[15]https://lightgbm.readthedocs.io/en/latest/pythonapi/lightgbm.LGBMClassifier.html

[16]https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC

[17]https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[18]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

- XGBoost classifier[19]

4.4.5.2 Repeated Stratified *k*-fold Cross-validation

The fundamental idea behind supervised learning is that a machine learning algorithm 'learns' from data and then uses the learned model to predict unseen data. Typically, practitioners split their data into training and test sets. The training set trains the model in isolation from the test set, which then evaluates the model's performance. Since the model did not access the test set during training, we consider the test set 'unseen data.' However, this approach can lead to a model that does not generalise well, as it is possible to overfit during training without a chance to correct it. Overfitting occurs when a model learns the training data patterns too well, resulting in poor performance on unseen data [Yang et al., 2022].

Cross-Validation[20] addresses this issue by improving generalisation, the ability of a classifier to perform well across various inputs. In cross-validation, the data is split into *k* subsets (folds), and the classifier is trained iteratively on *k-1* folds, using the remaining fold as a test set. This process repeats until each fold is used for training and testing. For example, given a dataset with 200 observations and *k* set to 10, 10-fold cross-validation splits the data into *ten* folds of 20 observations each. It trains the model on 180 observations and tests it on 20, repeating this process with different folds until all folds have been used. This yields a more reliable result based on the average of the iterations, provided there is no data leakage between the training and test sets.

Stratified *K*-fold Cross-Validation[21] ensures that each fold maintains the same class distribution between vulnerable and non-vulnerable methods, which is crucial given our class distribution of 1.48 : 98.52 as shown in Table 4.11. Traditional *k*-fold cross-validation does not account for class distributions, which could lead to inaccurate results due to disproportionate class representation in some folds.

[19]https://xgboost.readthedocs.io/en/stable/python/index.html
[20]https://scikit-learn.org/stable/modules/cross_validation.html
[21]https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html

Repeated Stratified k-fold Cross-Validation[22] repeats the stratified k-fold cross-validation $\boldsymbol{n}$ times. For example, with k set to 10 and n set to 3, it performs 10-fold cross-validation three times, reporting the average performance metrics over the repetitions. This averaged figure is more reliable than a single cross-validation or train-test split figure.

To ensure reliable classification results, we performed repeated stratified k-fold cross-validation with k and $\boldsymbol{n}$ both set to 10. Thus, we repeated the stratified 10-fold cross-validation ten times for each of our ten classifiers and reported the average performance metrics obtained.

4.4.5.3 Evaluation Metrics

Model evaluation[23] is crucial in classification tasks to determine a classifier's performance and suitability for a given task. We evaluated our classifiers using ground truth data (see Subsubsections 4.4.2.2 and 4.4.3.4) and three metrics: precision, recall, and F1 score.

Binary classification involves two classes: positive and negative. The positive class is usually the class of interest, which, in our case, is the vulnerable method. The negative class is the opposite of the positive class, which, in our case, is the non-vulnerable method. We used the following terms to describe the classification results:

- True Positive (TP): Correctly predicts an observation as positive (e.g., identifying a vulnerable method as vulnerable).
- True Negative (TN): Correctly predicts an observation as negative (e.g., identifying a non-vulnerable method as non-vulnerable).
- False Positive (FP): Incorrectly predicts an observation as positive (e.g., identifying a non-vulnerable method as vulnerable).
- False Negative (FN): Incorrectly predicts an observation as negative (e.g., identifying a vulnerable method as non-vulnerable).

[22] `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RepeatedStratifiedKFold.html`
[23] `https://scikit-learn.org/stable/modules/model_evaluation.html`

Precision measures the proportion of true positives among all predicted positives:

$$\text{Precision} = \frac{TP}{TP+FP}$$

Recall measures the proportion of true positives among all actual positives:

$$\text{Recall} = \frac{TP}{TP+FN}$$

F1 score is the harmonic mean of precision and recall, balancing the two:

$$\text{F1 score} = 2 * \frac{(Precision * Recall)}{(Precision + Recall)}$$

The F1 score is particularly suitable for imbalanced datasets [Al-Azani and El-Alfy, 2017] like ours. We chose it as the primary evaluation metric for the following reasons:

1. It is simple to understand and interpret.
2. It is widely used in the machine learning community, facilitating comparison with other studies.
3. It places less emphasis on true negatives, which are less relevant in our context, given that we are more interested in identifying vulnerable methods than non-vulnerable ones.

4.4.6 Approach to Research Question 1

The experiments in this chapter addressed the first research question of this thesis: *How well does the information retrieval-driven software vulnerability prediction technique perform on a single, multi-release software system dataset for token-based source code representations?*

To address this research question, we aimed to achieve the following objectives:

1. Identify the most suitable classifier for information retrieval-driven, token-based, method-level vulnerability prediction.

2. Determine the best-performing combination of token-based software metrics for vulnerability prediction.

3. Evaluate the impact of hyperparameter tuning on the performance of the best-performing classifier.

4.4.6.1 Objective 1

Identify the most suitable classifier for information retrieval-driven token-based method-level vulnerability prediction.

We used precision and recall to evaluate classifier performance and the F1 score to balance these two metrics for the first objective. As explained in Subsubsection 4.4.5.3, the F1 score combines precision and recall into a single value. Thus, we used the metric as the primary evaluation criterion to determine the best classifier for our prediction task.

4.4.6.2 Objective 2

Identify the best-performing token-based software metrics combination for vulnerability prediction.

The second objective involved analysing the best-performing combination of software metrics for vulnerability prediction from all sixteen metrics.

Building an interpretable model in machine learning requires understanding how different features affect its performance. Therefore, feature selection is crucial, as the chosen features directly impact model performance. This is a sentiment shared by Shivaji et al. [2009] and Theisen and Williams [2020].

With sixteen software metrics to consider, we used a feature selection technique to identify the best combination for our classification models.

Before feature selection, we conducted a correlation analysis to assess the relationship between the metrics and the ground truth data. This analysis helps clarify how each metric correlates with the target variable (ground truth) or other metrics. A high correlation with the target variable indicates essential features, while a high inter-metric correlation might lead to redundancy.

In machine learning, correlation analysis helps identify redundant features that could negatively impact model performance. However, evaluating all possible combinations of the sixteen metrics is impractical due to the exponential increase in combinations. For instance, evaluating five metrics requires checking 31 combinations, while sixteen require checking 65,535 combinations, making this approach time-prohibitive. Thus, the time complexity of the brute-force approach is $O(2^n)$, where n is the number of metrics, which is exponential and, therefore, time-prohibitive. in contrast, the Sequential Forward Selection algorithm has a time complexity of $O(n^2)$, making it more efficient.

We employed Sequential Feature Selection[24], which selects features sequentially to find the best-performing combination. We identified the best-performing combination for every n number of metrics, where n ranges from 1 to 15, in addition to the baseline model that uses all 16 metrics. We then compared these to identify the best-performing combination.

Sequential Feature Selection is a greedy search algorithm that aims to identify the best-k-performing feature combination out of n features for a given machine learning model, where $k < n$ and k are specified *a priori*. It has two basic variants: Forward Selection and Backward Selection. Forward Selection starts with an empty set of features and adds one feature at a time until it reaches k features. Backward Selection starts with all features and removes one feature at a time until it reaches k features. We used Forward Selection, the default variant in the scikit-learn library.

To illustrate, suppose we have a dataset with four features ($n = 4$) and want to identify the best three-feature combination using Sequential Feature Selection (Forward Selection) to achieve the highest F1 score. The algorithm proceeds as follows:

1. Start with an empty set of features.

2. Train a model using each feature in the dataset and evaluate the F1 score of each model ($k = 1$).

[24]`https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SequentialFeatureSelector.html`

3. Identify the feature that yields the best performance.

4. Train a model using the feature identified in step 3 and each remaining feature, then evaluate the F1 score of each model ($k = 2$).

5. Identify the best-performing feature duo.

6. Train a model using the feature duo identified in step 5 and each remaining feature, then evaluate the F1 score of each model ($k = 3$).

7. Identify the best-performing feature trio.

8. Return the set of features identified in step 7.

In a more complex scenario, to identify the best single feature, the best-performing feature duo, the best-performing feature trio, and the baseline performance using all four features, we would need to run the algorithm four times. The first three runs would have k equal to 1, 2, and 3, respectively, and the fourth would involve training and evaluating a baseline model using all four features. Algorithms 1 and 2 present the pseudocode for this more complex scenario used in our experiment.

For simplicity, we used a small hypothetical dataset with four features in our illustration, but the same approach applies to our sixteen-feature dataset. The algorithm runs sixteen times for each k value, where k ranges from 1 to 15, plus the baseline model training using all sixteen features.

Algorithm 1 declares the independent variables, X, on line 2, the number of features in the dataset, n, on line 3, and the dependent variable, y, on line 4. It then declares the cross-validation strategy, cv, on line 5. The algorithm iterates through the number of features in the dataset, n, on line 6 and calls the *run_classification* procedure on line 8. The call to the *run_classification* procedure includes a Boolean flag, $is\_baseline$, on line 7 to indicate whether the current iteration is for training and evaluating a baseline model. $is\_baseline$ is set to true if k is equal to n, which indicates that the current iteration is for training and evaluating a baseline model using all features.

Algorithm 1 Sequential Feature Selection and Classification (Part I of II)

```
1: features ← features_in_dataset          ▷ List containing CSV column names.

2: X ← independent_variables               ▷ CSV column data for features.
3: n ← number_of_features
4: y ← dependent_variable                  ▷ Binary vulnerability status label: ‘0’ or ‘1’

5: cv ← cross_validation_object            ▷ Repeated Stratified K-Fold.

6: for k ← 1, n do                         ▷ ‘k′’: number of features to select.
7:     is_baseline ← k = n
8:     run_classification(X, y, is_baseline, k, cv)
9: end for
```

Algorithm 2 Sequential Feature Selection and Classification (Part II of II)

```
1: procedure RUN_CLASSIFICATION(X, y, is_baseline, k, cv)

2:     classifiers ← classifier_objects_list

3:     for classifier ∈ classifiers do

4:         pl ← create_pipeline()          ▷ Add MinMaxScaler, SMOTE & classifier.
5:         data_map ← empty_map

6:         if is_baseline then             ▷ Run baseline classification.
7:             data_map ← train_and_evaluate_pipeline(pl, X, y, cv)
8:         else                            ▷ Run classification with feature selection.
9:             scoring ← “F1 score”
10:            best_k_features ← sequential_feature_selector(pl, k, scoring, cv)
11:            data_map ← train_and_evaluate_pipeline(pl, best_k_features, y, cv)
12:        end if

13:        plot_data(data_map)             ▷ Plot F1 score, Precision and Recall data.

14:    end for
15: end procedure
```

The *run_classification* procedure is declared in Algorithm 2. It takes the independent variables X, the dependent variable y, the Boolean flag $is\_baseline$, the number of features to select k, and the cross-validation strategy cv as parameters. It iterates through all nominated classifiers on line 3 and calls the *create_pipeline* procedure on line 4. The *create_pipeline* procedure creates a pipeline pl alongside pipeline steps, comprising a MinMaxScaler (Feature Scaling) object, a SMOTE object, and an estimator object, i.e, the classifier.

If the current iteration is for training and evaluating a baseline model, the *run_classification* procedure calls the *train_and_evaluate_pipeline* procedure on line 7, which trains and evaluates the classifier in the pipeline using *all* features. It then calls the *plot_data* procedure on line 13 to plot the current classifier's F1 score, Precision, and Recall data.

Suppose the current iteration is not for training and evaluating a baseline model. In that case, the *run_classification* procedure calls the *sequential_feature_selector* procedure on line 10, a scikit-learn implementation of the Sequential Feature Selection algorithm, to identify the best-performing features for the current classifier. After that, it calls *train_and_evaluate_pipeline* on line 11, which trains and evaluates the classifier in the pipeline using the best-performing features identified by Sequential Feature Selection. Note that the best-performing features $best\_k\_features$ are passed on line 11, unlike the X variable used for the baseline model on line 7.

Finally, the *run_classification* procedure calls the *plot_data* procedure on line 13 to plot the current classifier's F1 score, Precision, and Recall data. The outputs from *plot_data* then inform the best-performing software metrics combination for each n number of metrics, addressing this objective.

4.4.6.3 Objective 3

Evaluate the impact of hyperparameter tuning on the performance of the best-performing classifier.

For the third objective, we conducted hyperparameter tuning to determine its impact on the predictive performance of the best classifier.

Hyperparameter tuning is essential in machine learning as it involves selecting the optimal hyperparameters for a model. The model does not learn directly from hyperparameters; instead, the hyperparameters control the learning process, such as the number of trees in a random forest classifier. Hyperparameter tuning is crucial because it can significantly affect a model's performance. For instance, a random forest classifier with 100 trees might outperform one with 50 trees, but this is not always the case. Therefore, it is often necessary to tune hyperparameters to find the best combination for a given model.

Two popular hyperparameter tuning techniques are Grid Search and Random Search. We used Grid Search[25] in our experiment. Grid Search exhaustively searches a manually specified subset of hyperparameter values and selects the best-performing combination. Although computationally expensive, it is straightforward and effective.

We combined Grid Search with Repeated Stratified *k*-fold cross-validation to tune the hyperparameters of our best-performing classifier using the *best-performing* metrics combination. The tuning process involved only the best-performing software metrics combination identified for the best-performing classifier in the previous objective, not all sixteen metrics.

The hyperparameter tuning process concludes our methodology for addressing Research Question 1. The following section presents and discusses our experimental results.

[25]`https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html`

4.5 Results

This section presents our experimental results, organised according to the objectives outlined in Subsection 4.4.6.

4.5.1 Objective 1 Results

Identify the most suitable classifier for information retrieval-driven token-based method-level vulnerability prediction.

This objective aimed to identify the best-performing machine learning binary classifier among the nominated classifiers in terms of predictive performance.

4.5.1.1 Evaluation Metrics Trend Analysis

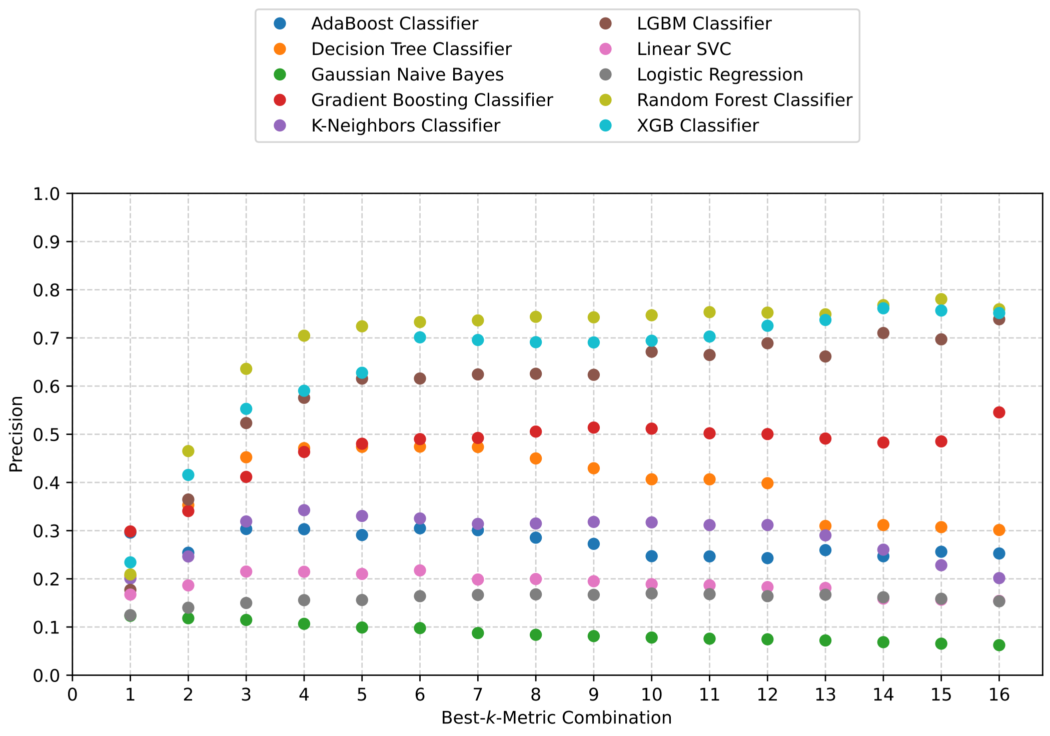


Figure 4.10: Precision Trend across all Best-k-Performing Metrics Combinations

Figure 4.10 shows the precision trend across all best-k-performing metrics combinations for each classifier, with the highest precision nearly 0.80, achieved by the Random Forest classifier at k = 15. The lowest precision is around 0.07, attained by the Gaussian Naïve Bayes classifier at k = 16.

The figure indicates that the Random Forest classifier achieved the highest precision and maintained the best precision trend across most best-k-performing metrics combinations. Additionally, the LGBM and XGB classifiers mostly showed above-average precision trends in most best-k-performing metrics combinations, with the Gradient Boosting classifier generally following the average precision trend and achieving its best above-average precision at $k = 16$. The remaining classifiers showed trends with below-average precision.

The Gaussian Naïve Bayes, Logistic Regression, and Linear Support Vector classifiers performed the worst in terms of precision. The Gaussian Naïve Bayes consistently achieved the lowest precision across all best-k-performing metrics combinations. Similarly, the Linear Support Vector classifier and Logistic Regression had low precision across most best-k-performing metrics combinations.

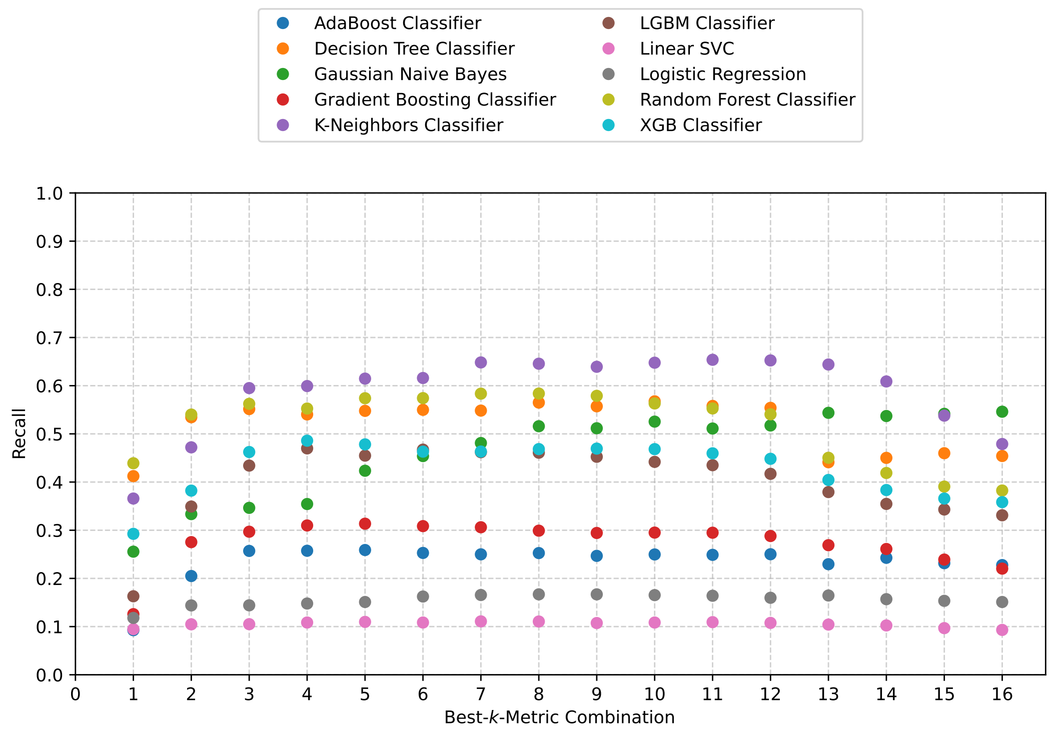


Figure 4.11: Recall Trend across all Best-k-Performing Metrics Combinations

Figure 4.11 shows the recall trend across all best-k-performing metrics combinations for each classifier, with the highest recall of approximately 0.65 achieved by the K-Nearest Neighbors classifier at $k = 11$. The lowest recall is approximately 0.09, attained by the Linear Support Vector classifier at $k = 16$.

The figure indicates that the K-Nearest Neighbors classifier achieved the highest recall and maintained the best recall trend across most best-k-performing metrics combinations. The Random Forest, Decision Tree, and Gaussian Naïve Bayes classifiers mainly showed above-average recall trends in most best-k-performing metrics combinations. The remaining classifiers showed below-average recall trends.

The worst-performing classifiers in terms of recall were the Linear Support Vector classifier, Logistic Regression, and AdaBoost classifier. The Linear Support Vector classifier consistently achieved the lowest recall across all best-k-performing metrics combinations. Similarly, the Logistic Regression and AdaBoost classifier had low recall across most best-k-performing metrics combinations.

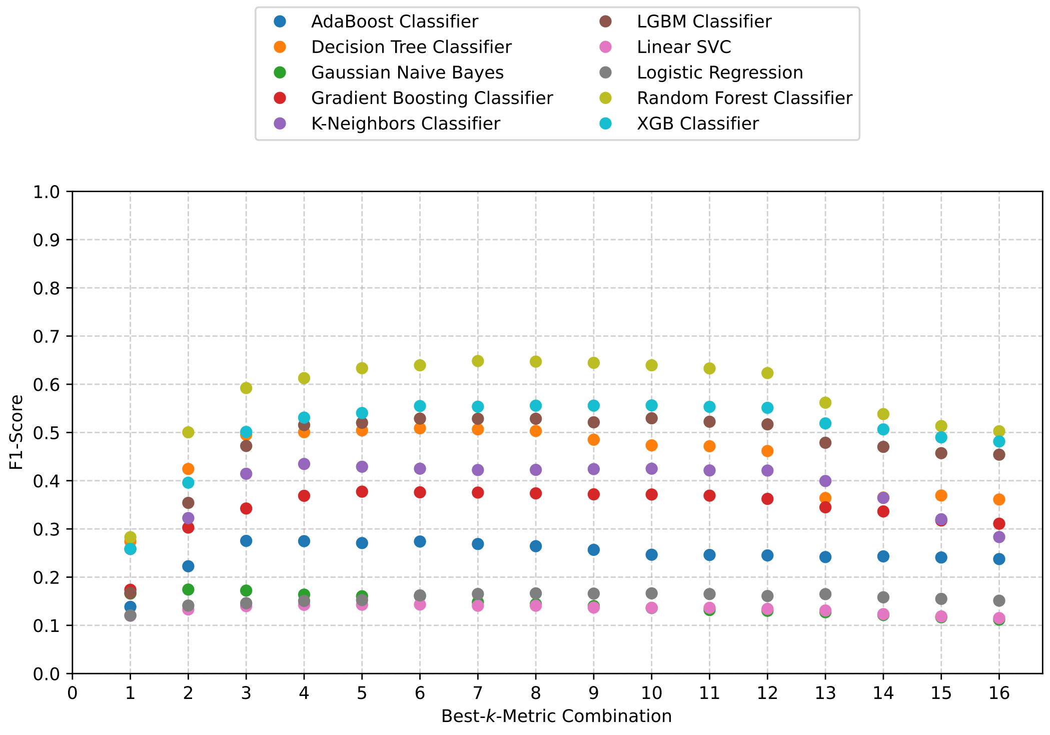


Figure 4.12: F1 score Trend across all Best-k-Performing Metrics Combinations

Figure 4.12 shows the F1 score trend across all best-k-performing metrics combinations for each classifier, with the highest F1 score of approximately 0.65 achieved by the Random Forest classifier at $k = 7$.

The figure indicates that the Random Forest classifier consistently achieves the highest F1 score across all best-*k*-performing metrics combinations and the baseline model.

It also shows that the XGB and LGBM classifiers achieved above-average F1 score trends in most best-*k*-performing metrics combinations. The Decision Tree classifier occasionally appeared along the average F1 score trend from $k = 4$ to $k = 8$.

The worst-performing classifiers in terms of F1 score were the Linear Support Vector classifier, Gaussian Naïve Bayes, and Logistic Regression. The Linear Support Vector classifier and Gaussian Naïve Bayes showed similar F1 score trends, while Logistic Regression performed slightly better but was still significantly below average.

Regarding the first objective, the Random Forest classifier achieved the best predictive performance, with the highest F1 score among all best-k-performing metrics combinations.

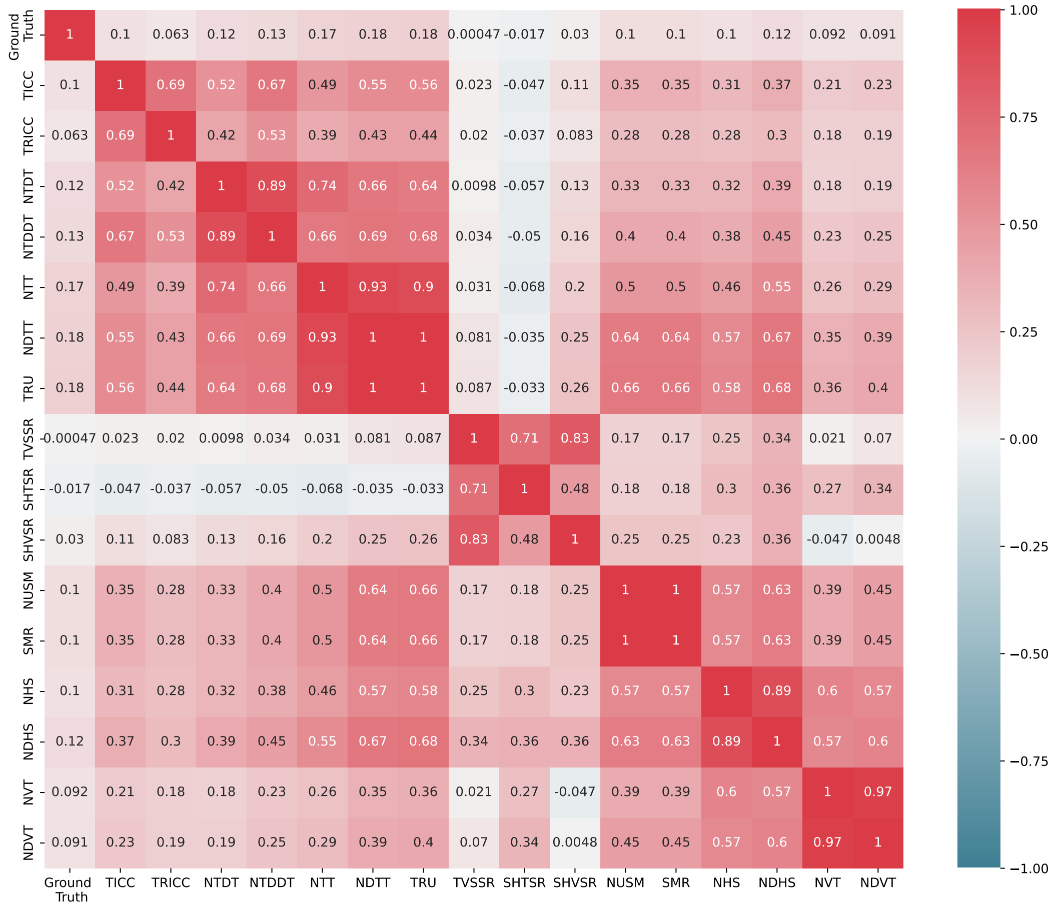
4.5.2 Objective 2 Results

Identify the best-performing token-based software metrics combination for vulnerability prediction.

This objective aimed to identify the best-performing combination of software metrics for vulnerability prediction from all sixteen metrics.

4.5.2.1 Metrics Correlation Analysis

Figure 4.13 shows the correlation matrix of all metrics and the ground truth. The figure displays the Pearson Correlation Coefficient for the ground truth in the first row and column, with the remaining metrics in the other rows and columns. The matrix is organised into two levels of groupings. First, the rows and columns are grouped into ground truth, hit-independent, and hit-dependent metrics. Then, the latter two groups are clustered according to their respective metrics. The Pearson Correlation Coefficient measures the linear correlation between two variables, ranging from '-1' (strong negative correlation) to '1' (strong positive correlation), with



	Ground Truth	TICC	TRICC	NTDT	NTDDT	NTT	NDTT	TRU	TVSSR	SHTSR	SHVSR	NUSM	SMR	NHS	NDHS	NVT	NDVT
Ground Truth	1	0.1	0.063	0.12	0.13	0.17	0.18	0.18	0.00047	-0.017	0.03	0.1	0.1	0.1	0.12	0.092	0.091
TICC	0.1	1	0.69	0.52	0.67	0.49	0.55	0.56	0.023	-0.047	0.11	0.35	0.35	0.31	0.37	0.21	0.23
TRICC	0.063	0.69	1	0.42	0.53	0.39	0.43	0.44	0.02	-0.037	0.083	0.28	0.28	0.28	0.3	0.18	0.19
NTDT	0.12	0.52	0.42	1	0.89	0.74	0.66	0.64	0.0098	-0.057	0.13	0.33	0.33	0.32	0.39	0.18	0.19
NTDDT	0.13	0.67	0.53	0.89	1	0.66	0.69	0.68	0.034	-0.05	0.16	0.4	0.4	0.38	0.45	0.23	0.25
NTT	0.17	0.49	0.39	0.74	0.66	1	0.93	0.9	0.031	-0.068	0.2	0.5	0.5	0.46	0.55	0.26	0.29
NDTT	0.18	0.55	0.43	0.66	0.69	0.93	1	1	0.081	-0.035	0.25	0.64	0.64	0.57	0.67	0.35	0.39
TRU	0.18	0.56	0.44	0.64	0.68	0.9	1	1	0.087	-0.033	0.26	0.66	0.66	0.58	0.68	0.36	0.4
TVSSR	-0.00047	0.023	0.02	0.0098	0.034	0.031	0.081	0.087	1	0.71	0.83	0.17	0.17	0.25	0.34	0.021	0.07
SHTSR	-0.017	-0.047	-0.037	-0.057	-0.05	-0.068	-0.035	-0.033	0.71	1	0.48	0.18	0.18	0.3	0.36	0.27	0.34
SHVSR	0.03	0.11	0.083	0.13	0.16	0.2	0.25	0.26	0.83	0.48	1	0.25	0.25	0.23	0.36	-0.047	0.0048
NUSM	0.1	0.35	0.28	0.33	0.4	0.5	0.64	0.66	0.17	0.18	0.25	1	1	0.57	0.63	0.39	0.45
SMR	0.1	0.35	0.28	0.33	0.4	0.5	0.64	0.66	0.17	0.18	0.25	1	1	0.57	0.63	0.39	0.45
NHS	0.1	0.31	0.28	0.32	0.38	0.46	0.57	0.58	0.25	0.3	0.23	0.57	0.57	1	0.89	0.6	0.57
NDHS	0.12	0.37	0.3	0.39	0.45	0.55	0.67	0.68	0.34	0.36	0.36	0.63	0.63	0.89	1	0.57	0.6
NVT	0.092	0.21	0.18	0.18	0.23	0.26	0.35	0.36	0.021	0.27	-0.047	0.39	0.39	0.6	0.57	1	0.97
NDVT	0.091	0.23	0.19	0.19	0.25	0.29	0.39	0.4	0.07	0.34	0.0048	0.45	0.45	0.57	0.6	0.97	1

Figure 4.13: Correlation Matrix of all Metrics + Ground Truth

'0' indicating no correlation. For interpretation, we categorise the values in the correlation matrix as shown in Table 4.13.

Table 4.13: Pearson Correlation Coefficient Value Bands

Band No.	Band	Description
1	0.8 – 1	Very strong positive correlation
2	0.6 – 0.8	Strong positive correlation
3	0.4 – 0.6	Moderate positive correlation
4	0.2 – 0.4	Weak positive correlation
5	0 – 0.2	Very weak positive correlation
6	0	No correlation
7	-0.2 – 0	Very weak negative correlation
8	-0.4 – -0.2	Weak negative correlation
9	-0.6 – -0.4	Moderate negative correlation
10	-0.8 – -0.6	Strong negative correlation
11	-1 – -0.8	Very strong negative correlation

The figure includes a scale on the right side of the matrix, featuring a diverging colour scheme. The red-toned area (toward the top) indicates a strong positive correlation, the white area (in the middle) indicates no correlation, and the teal-toned area (toward the bottom) indicates a strong negative correlation.

We present the main observations from the correlation matrix below. For reference, the full names of the metrics are in Table 4.1 and Table 4.4.

1. Some cells in Band No. 1 feature a value of ‘1’ and values close to ‘1’, indicating a strong positive correlation between certain metrics. Examples include the correlation between NUSM & SMR, as well as between several metrics and their distinct counterparts, such as NTT & NDTT. These high correlations align with expectations as the metrics are conceptually similar. For instance, SMR is a relative counterpart of NUSM, & NDTT is a distinct counterpart of NTT. Such correlations suggest possible redundancy between metrics, which may or may not adversely affect model performance.

2. Regarding negative correlation, the highest values fall within Band No. 8. For example, the highest observed negative correlation is between TRICC & SHTSR, around -0.037. TRICC represents the relative code churn of a method, while SHTSR represents the ratio of the number of hit shingles to the total number of shingles in a method. The negative correlation indicates that the hit shingles ratio decreases as the relative code churn of a method increases. However, these negative correlations are not strong, indicating caution when interpreting these values.

3. The NTT & the NVT are conceptually similar metrics, but they cater to different datasets in our analysis, i.e., the target software system and the vulnerability dataset. As such, these metrics exhibit a weak positive correlation at 0.23 (Band No. 4) because each metric focuses on a different dataset. However, this positive correlation improves to 0.48 (Band No. 3) when considering hits, as observed between SHTSR & SHVSR.

4. SHTSR assesses a target software system's method, while SHVSR assesses its matching method in the vulnerability dataset. TVSSR considers the target software system and its matching method in the vulnerability dataset. Thus, TVSSR shares a closer bond with SHTSR & SHVSR, resulting in higher correlations of 0.71 (Band No. 2) and 0.83 (Band No. 1), respectively, compared to the correlation between SHTSR & SHVSR at 0.48 (Band No. 3).

5. While SMR is the relative version of NUSM, their relationship is more stable than that between TRICC & TICC. This stability is due to the fact that the denominator used in calculating SMR, i.e., the total number of matches in the vulnerability dataset, remains constant, whereas the denominator for TRICC, i.e., the total number of target software system releases, changes. Consequently, the correlation between NUSM & SMR is higher than that between TRICC & TICC.

6. The ground truth does not exhibit a strong correlation with any single metric, with the highest correlation of 0.18 (Band No. 5) observed between the ground truth and NDTT & TRU. This lack of strong correlation suggests that despite the redundancy indicated by the correlation matrix, some metrics are crucial for classifying the ground truth effectively, as shown by the above-average performance of at least four classifiers.

7. The low correlation values between the ground truth and any of the sixteen metrics suggest that software vulnerability prediction is a challenging task. Since no single metric significantly correlates with the ground truth, meaningful results likely arise from a combination of metrics. The low correlation values of the ground truth highlight one of the main challenges in software vulnerability prediction research.

Conceptually, as explained in Subsubsection 4.4.6.2, the correlation matrix helps identify and exclude redundant metrics to improve model performance. However, in our case, excluding seemingly redundant metrics such as SMR and the 'distinct' metric variants (NTDDT, NDHS, NDVT, and NDT) did not improve performance; in many cases, it actually worsened performance across different classifiers, albeit marginally.

We attribute this to feature interactions, where combining features yields results that differ from the sum of the individual feature results. Although the correlation matrix may flag specific metrics as redundant, these features can still interact synergistically, aiding the successful prediction of the ground truth.

Therefore, we included all metrics in our analysis to avoid inadvertently excluding those that may be beneficial. We opted to apply feature selection, an essential activity in prediction analysis, as supported by Shivaji et al. [2009]'s work. Thus, we relied on the Sequential Feature Selection algorithm to identify the best-performing metrics combinations. For instance, SMR, NTDDT, NDHS, NDVT, and NDTT may seem redundant due to their conceptual similarity to other metrics—SMR is the relative variant of NUSM, NTDDT is the distinct variant of NTDT, NDHS is the distinct variant of NHS, NDVT is the distinct variant of NVT, and NDTT is the distinct variant of NTT. However, the Sequential Feature Selection algorithm identified these metrics as part of the best-performing combinations, justifying their inclusion in our analysis.

4.5.2.2 Classifier Performance Analysis

Table 4.14: Best Performance Per Classifier (Sorted by F1 score)

Classifier	**Best k**	**Precision**	**Recall**	**F1 score**
Random Forest classifier	7	0.73635	0.58345	0.64821
XGBoost classifier	10	0.69415	0.46830	0.55622
LightGBM classifier	10	0.67129	0.44188	0.52947
Decision Tree classifier	6	0.47433	0.54988	0.50878
K-Nearest Neighbors classifier	4	0.34247	0.59923	0.43471
Gradient Boosting classifier	5	0.48036	0.31356	0.37731
AdaBoost classifier	3	0.30337	0.25707	0.27521
Gaussian Naive Bayes	2	0.11818	0.33347	0.17425
Logistic Regression	8	0.16774	0.16693	0.16634
Linear Support Vector classifier	6	0.21746	0.10846	0.14320

Table 4.14 shows the best performance per classifier, sorted by descending F1 score, our preferred metric for identifying the best-performing classifier, as it

balances precision and recall. The ‘Best *k*’ column indicates the number of features that achieved the performance.

The table shows that the Random Forest classifier had the highest F1 score. The XGBoost, LightGBM, and Decision Tree classifiers also performed well, with above-average F1 scores.

Conversely, the Linear SVC, Logistic Regression, and Gaussian Naïve Bayes classifiers were among the worst performers in terms of F1 score.

4.5.2.3 Metrics Combination Analysis

Table 4.15: Best Metrics Combination Per Classifier

Metric	**Classifiers**									
	AB	***DT***	***GNB***	***GB***	***KN***	***LGBM***	***LSVC***	***LG***	***RF***	***XGB***
Hit-Independent										
NTT	✓		✓	✓	✓	✓		✓	✓	✓
NDTT	✓			✓		✓	✓			✓
TICC							✓			
TRICC							✓	✓		
NTDT								✓		
NTDDT										
TRU							✓	✓		
Hit-Dependent										
NHS		✓				✓	✓		✓	✓
NDHS		✓					✓			
NVT		✓			✓	✓		✓	✓	✓
NDVT	✓	✓		✓	✓	✓		✓	✓	✓
TVSSR			✓	✓		✓		✓		✓
SHTSR						✓		✓		✓
SHVSR		✓				✓			✓	✓
NUSM					✓	✓			✓	✓
SMR		✓		✓		✓			✓	✓

Table 4.15 displays the optimal metrics combinations for each classifier, categorised into hit-independent and hit-dependent groups.

The table offers several insights. Notably, it shows that hit-dependent metrics are more crucial for classifiers than hit-independent metrics, as most classifiers include more hit-dependent metrics in their best-performing combinations. This is expected since these metrics capture vulnerable code patterns in the vulnerability dataset. We define the best-performing classifiers as those achieving average or above-average F1 scores, as reported in Table 4.14.

Additionally, the table highlights the importance of individual metrics. For instance, the NTT and NDVT metrics appear eight times in the best-performing combinations, signifying their importance. In contrast, the TICC and NTDT metrics appear only once, indicating lesser significance.

We also note the poor performance of code churn-related metrics, TICC and TRICC, in the analysis. This aligns with the correlation matrix findings, showing weak correlations between churn metrics, ground truth, and other metrics. As shown in Figure 4.13, the correlation values for churn metrics are primarily located in the lighter areas, indicating weaker correlations compared to most hit-independent metrics.

The TRICC metric appears only twice for the Linear SVC and Logistic Regression classifiers, both of which are poor performers, as shown in Table 4.14. Similarly, the TICC metric appears only once for the Linear SVC, another poor performer.

Thus, for the second objective, the optimal software metrics combination for vulnerability prediction includes those used by the best-performing classifier, the Random Forest classifier. These metrics are NDVT, NHS, NTT, NUSM, NVT, SHVSR, and SMR.

4.5.3 Objective 3 Results

Evaluate the impact of hyperparameter tuning on the performance of the best-performing classifier.

This objective aimed to evaluate the impact of hyperparameter tuning on the performance of the best-performing classifier identified in the first objective.

4.5.3.1 Parameter Grid and Best Hyperparameter Values

Table 4.16 presents the parameter grid details and the best hyperparameter values for the Random Forest classifier, which was identified as the best-performing classifier in the first objective. The first two columns show the hyperparameters and values

Table 4.16: Parameter Grid and Best Hyperparameter Values

Parameter Grid		
Hyperparameter	***Values***	***Best Value***
bootstrap	True*, False	False
max_depth	None*, 10, 20	None
max_features	'auto', 'sqrt'*	auto
min_samples_leaf	1*, 2, 4	1
min_samples_split	2*, 5, 10	2
n_estimators	100*, 200, 300	100

we used to tune the Random Forest classifier. The asterisk (*) denotes the default value in the scikit-learn implementation. We focused on six hyperparameters due to time and computational resource constraints.

The last column shows the best-performing hyperparameter values determined by our Grid Search technique (see Subsubsection 4.4.6.3). We tuned the hyperparameters using the best-performing metrics combination identified in the second objective: NDVT, NHS, NTT, NUSM, NVT, SHVSR, and SMR.

4.5.3.2 Hyperparameter Tuning Results

Table 4.17: Pre-and-Post-Hyperparameter Tuning Results for Random Forest Classifier

Metric	**Before**	**After**	$\Delta\%$
Precision	0.73635	0.73472	-0.22
Recall	0.58345	0.59667	2.23
F1 score	0.64821	0.65741	1.42

Table 4.17 presents the pre- and post-hyperparameter tuning results for the Random Forest classifier. Tuning the hyperparameters resulted in changes to the classifier's performance metrics. Precision decreased by 0.22%, recall increased by 2.23%, and the F1 score increased by 1.42%.

To address the third objective, hyperparameter tuning had a mixed impact on the performance of the best-performing classifier, the Random Forest classifier. However, the overall impact was positive, as indicated by the increase in the F1 score metric, our preferred measure for evaluating classifier performance.

4.6 Discussion

This chapter investigated a novel approach to vulnerability prediction using token-based representations and information retrieval techniques. We conducted an empirical analysis using the Apache Tomcat 7 software system, encompassing 76 releases and over 20,000 vulnerable code samples from the NIST Software Assurance Reference Dataset. The objective was to assess the effectiveness of this method for predicting method-level vulnerabilities. The Random Forest classifier emerged as the top-performing model, achieving a precision of 0.73, a recall of 0.60, and an F1 score of 0.66 after hyperparameter tuning. These results demonstrate the promise of information retrieval-driven techniques for practical vulnerability prediction.

The following subsections discuss the study's findings, key observations, and implications and offer recommendations for practitioners and researchers.

4.6.1 The Importance of Interpretability in Prediction Models

An interpretable prediction model is crucial for understanding the relationship between predictors (independent variables) and the target variable (dependent variable). Figure 4.12 illustrates a vital insight: the inclusion of more predictors does not necessarily lead to more accurate predictions, a finding implied by Shivaji et al. [2009]'s study. The figure shows that the F1 score trend improves with most classifiers from $k = 2$ and peaks (and often plateaus) at any value before or after $k = 8$. It then gradually declines as k approaches 16. This trend indicates that while adding more predictors can enhance prediction performance, there is a point of diminishing returns. Beyond this point, additional predictors may not significantly improve performance and may even degrade it. Therefore, aiming for interpretability in prediction models is vital. This approach can help identify the optimal balance between the number of predictors and prediction performance.

4.6.2 The Significance of Vulnerable Code Patterns

The best-performing metrics combination—NDVT, NHS, NTT, NUSM, NVT, SHVSR, and SMR—primarily includes hit-dependent metrics. The only hit-independent metric in the combination is NTT. Since hit-dependent metrics fundamentally measure code element similarity to vulnerable code patterns, we deduce that code patterns, especially those consistent with vulnerable code, are crucial for predicting vulnerabilities. Thus, the similarity measure is the most significant predictor of vulnerabilities.

Another perspective that showcases the validity of the significance of vulnerable code pattern deduction is in the context of vulnerability types. A few related works in the literature focused on specific vulnerability types, such as the improper use of programming language features, API misuse, SQL injection, cross-site scripting, operating system command injection, and buffer overflow. These vulnerability types are characterised by specific code patterns that, although they may vary in complexity, are fundamentally similar across instances of the same vulnerability type and thus finite. This similarity in code patterns across instances of the same vulnerability type is a key factor in the success of vulnerability prediction models, as we will highlight by synthesising the findings from the works of Guo et al. [2023], Rabheru et al. [2022], and Wu et al. [2023].

Guo et al. [2023] developed VulExplore, a novel vulnerability detection model that combines code metrics (CMs) with a composite neural network comprising a Convolutional Neural Network (CNN) and a Long Short-Term Memory (LSTM). The authors constructed a CM dataset from a publicly available code slice dataset containing four types of vulnerabilities in C/C++. They also introduced two additional software engineering-related CMs: maintainability index and average number of vulnerabilities per line. They designed a CNN-LSTM network to extract features from these CMs and learn deep representations of the code slices. The model was evaluated using k-fold cross-validation and compared with other tools and methods. The results demonstrated that VulExplore achieved high precision, recall, and F1 scores (over 80%) while reducing both false negative and false positive rates (under 20%). The authors claimed that their model outperformed existing tools and methods in terms of accuracy and coverage. They concluded that VulExplore is an effective and superior approach for vulnerability detection based on CMs. The critical similarity between their work and ours lies in their use of the SARD dataset and similar metrics concerned with code intricacy and size. However, the primary difference is that they employed deep learning techniques, while we used machine learning approaches. Additionally, our study addressed a broader range of vulnerability types, covering a wide range across our within- and mixed-project datasets, whereas their study focused on only four types.

Rabheru et al. [2022] developed a deep learning approach that combines Gated Recurrent Units (GRU) and Graph Convolutional Networks (GCN) for detecting PHP vulnerabilities. Their study aimed to develop a hybrid technique capable of capturing both syntactic and semantic information from PHP source code, enabling the accurate detection of SQLi, XSS, and OSCI vulnerabilities with strong generalisability. The authors introduced DeepTective, a deep learning model comprising two components: a GRU that processes token sequences and a GCN that operates on the source code's control flow graph. The model was evaluated against other tools using a synthetic dataset (SARD) and a realistic dataset (GIT) from GitHub. The results showed that DeepTective outperformed other tools on both datasets, achieving F1 scores of 99.92% on SARD and 88.12% on GIT. Additionally, DeepTective identified four novel vulnerabilities in deployed WordPress plugins. The study concluded that DeepTective is an effective and efficient vulnerability detection method that leverages the strengths of both GRU and GCN. Like our study, the study utilised the SARD dataset. However, they focused on PHP while we worked with Java. Additionally, they employed deep learning techniques, whereas we used machine learning approaches. Finally, similar to Guo et al. [2023], their study focused on four types of vulnerabilities, while we considered a broader range of vulnerabilities.

Wu et al. [2023] proposed a novel fine-grained code vulnerability detection model called SlicedLocator, which can predict vulnerabilities at both the program and statement levels. The study introduced a new code representation method, the Sliced Dependence Graph (SDG), which preserves rich interprocedural relationships while eliminating irrelevant statements. Additionally, the authors designed attention-based code embedding networks and a fusion model that combines

LSTM and GNN to capture the semantic and structural features of SDGs. SlicedLocator was evaluated using a large-scale C/C++ vulnerability dataset collected from CVE-fixes and SARD, covering 25 common vulnerabilities and 15 real-world software projects. The model was compared with other methods, including fine-grained approaches such as IVDETECT and LineVD, and coarse-grained methods like VulDeePecker, SySeVR, and Devign, using metrics such as Mean Average Precision (MAP), Recall, Normalized Discounted Cumulative Gain (nDCG), First Ranking (FR), and Average Ranking (AR). The results showed that SlicedLocator outperformed state-of-the-art vulnerability detection and localisation methods, particularly in localisation metrics, achieving a Macro F1 score of 0.6879. The study demonstrated that the SDG, code embedding networks, and LSTM-GNN could significantly enhance vulnerability localisation. It also revealed that the dual-grained training approach, which predicts vulnerabilities at both program and statement levels, improves detection performance. The study concluded that SlicedLocator is an effective fine-grained code vulnerability detection model that can assist security engineers in efficiently analysing and fixing vulnerabilities. The authors suggested future improvements, such as incorporating more types of code that can cause vulnerabilities, using advanced language models, and applying the model to other programming languages. This study is comparable to ours in terms of the F score and the use of the SARD dataset. However, they focused on C/C++, while we focused on Java. Additionally, they employed deep learning techniques, whereas we used machine learning approaches. Finally, they addressed 25 types of vulnerabilities, while we considered a broader range of vulnerabilities.

A critical observation from the three studies discussed above, which all employed the SARD vulnerability dataset, is that the first two studies, which limited their focus to a very few vulnerability types, reported higher performance. This suggests that the number of vulnerability types considered is inversely proportional to predictive performance. For example, Guo et al. [2023] and Rabheru et al. [2022] only considered four vulnerabilities and reported F1 scores exceeding 80% and 99.92%, respectively. Conversely, Wu et al. [2023]'s work, which addressed 25 vulnerability types, reported a lower F1 score of 0.6879. This trend suggests that a broader range of vulnerabilities correlates with the requirement for prediction models to recognise a more diverse set of vulnerable code patterns, which may be more challenging. On the other hand, a narrower focus on fewer vulnerability types may enable models to learn more specific and consistent code patterns, resulting in higher performance. This observation highlights the significance of identifying vulnerable code patterns in predicting vulnerabilities, as we have highlighted in our study.

Additionally, code patterns have a significant impact on the performance of prediction models on a broader scale in a dataset generalisability context. We will discuss this phenomenon extensively in Chapter 6.

4.6.3 The Token-Based Relative Instantaneous Code Churn Metric Design

The design of the TRICC metric could pose a threat to internal validity, depending on the analysis. In Subsubsection 4.3.1.4, we defined a method's TRICC as the ratio of its token representation change count to the total number of releases in which it appeared. Mathematically, we express it as the ratio of the TICC to N, where N is the total number of releases in which the method exists within the target software system.

This design approach encodes the 'future knowledge' of a method's evolution into the metric through N, allowing a machine learning classifier to learn from a method's history. This could be problematic in scenarios where future knowledge of the evolution of a software system's artefact leads to data leakage during machine learning analysis, a point also highlighted in Subsection 3.3.2 by Chowdhury et al. [2024]'s work.

We designed the TRICC metric to capture the entire evolution of a method over time through N at any point in its history rather than an instance of its evolution at a specific time. Our approach was based on the fact that we were more interested in quantifying each method's relative evolution history, as this is a more interpretable measure of a method's evolution.

Regardless, we point out that depending on the nature and objectives of the machine learning analysis, it may be necessary to make N instantaneous, where N represents the number of releases in which the method exists at a given point in time in the method's evolution history, rather than the total number of releases in which the method exists.

Nonetheless, our TRICC metric design choice does not affect the validity of our findings. As shown in Table 4.15, the TRICC metric is not part of the best-performing metrics combination for any of our best-performing classifiers with above-average F1 scores. It only appears in the best-performing metrics combination for the Linear SVC and Logistic Regression, which have some of the lowest F1 scores among the classifiers.

4.6.4 Feature Interactions: Synergism and Antagonism

Metrics can have synergistic or antagonistic effects when combined, as evidenced by the trends in Figures 4.10, 4.11, and 4.12.

For instance, Figure 4.10 shows that the precision trend for the XGB classifier dips after $k = 6$ before rising past its previous level at $k = 11$. This dip at $k = 6$ is due to adding a seventh metric to an already effective six-metric combination, causing an antagonistic effect. Continuous metric additions maintained this antagonistic effect until the eleventh metric reversed it. Similarly, in Figure 4.11, the recall trend for the Random Forest classifier dips after $k = 3$. It rises again after $k = 5$, indicating that the fourth metric had an antagonistic effect on recall, which the fifth metric reversed, creating a synergistic effect. Figure 4.12 also shows a dip between $k = 8$ and $k = 10$ for the LGBM classifier, suggesting an antagonistic effect of the ninth metric on the F1 score. These examples demonstrate that interactions between metrics can result in synergism or antagonism. Even though a metric may have a positive or negative impact on its own, its inclusion with others can result in different outcomes, which can be positive (synergistic) or adverse (antagonistic).

Sequential Feature Selection is designed to identify the best-performing k metrics combination at any point, selecting the top-performing metrics at each iteration. Ideally, this would result in a steady increase in performance as k increases until it reaches a peak, after which the trend would decline as k increases further. However, Figures 4.10, 4.11, and 4.12 show that while the trends generally have upward or downward trajectories, they are not always monotonically increasing or decreasing. This variation is a consequence of the synergistic or antagonistic interactions between metrics.

This complex interplay between metrics underscores the importance of considering feature interactions in prediction analysis and the need to employ feature selection techniques that can identify the best-performing metrics combinations.

4.6.5 Implications

The implications of this chapter are multifaceted. Our findings offer a pathway for software security practitioners to integrate advanced machine learning techniques into vulnerability prediction workflows. The Random Forest classifier has demonstrated effectiveness, an attribute it most likely possesses due to its ability to handle high-dimensional data and non-linear relationships. Also, the identified vital metrics provide a practical blueprint for developing information retrieval-driven predictive models.

For researchers, the study opens avenues for further exploration into the synergy between information retrieval and machine learning. Insights into feature interactions and the importance of comprehensive metric inclusion highlight areas that warrant further investigation. Additionally, the approach can be extended to other programming languages and software systems, potentially broadening its applicability.

4.6.6 Recommendations

Based on the findings and insights presented in this chapter, we offer several recommendations. Firstly, practitioners should incorporate token-based information retrieval techniques in their vulnerability prediction efforts. Model development should prioritise the identified key metrics (NDVT, NHS, NTT, NUSM, NVT, SHVSR, and SMR). Secondly, hyperparameter tuning should be an integral part of the model training process. Employing techniques such as grid search, combined with cross-validation, can significantly enhance model performance. Thirdly, data-related challenges in vulnerability prediction deserve more attention. Techniques such as transfer learning, which leverages knowledge from one dataset to another, could mitigate this challenge and improve model generalisation. Lastly, we recommend localising the vulnerability dataset to specific vulnerable code segments rather than entire methods. This granularity can provide classifiers with more precise information, potentially enhancing predictive performance.

4.7 Threats to Validity

In this section, we discuss the threats to the validity of our study. We categorise the threats into internal and external validity.

4.7.1 Internal Validity

Internal validity refers to the extent to which a study's design, execution, and analysis support the conclusions drawn from it.

4.7.1.1 Oversampling and Undersampling Techniques

We used SMOTE to oversample the minority ('vulnerable') class. However, various other techniques exist. For instance, some oversampling techniques include random oversampling [Mohammed et al., 2020], Adaptive Synthetic Sampling (AdaSyn) [He et al., 2008], and augmentation [Shorten and Khoshgoftaar, 2019]. Undersampling techniques include cluster-based undersampling [Zhang et al., 2010], Tomek Links [Tomek, 1976, Devi et al., 2017], and ensemble learning-based undersampling [Sarkar et al., 2020]. Changing the parameters or substituting SMOTE with these techniques may result in slightly different analysis outcomes, which can affect internal validity.

4.7.1.2 Ground Truth Estimation: Number of Affected Methods

We estimated the number of methods affected by the 261 fixed vulnerabilities in the ground truth data by assuming that a vulnerability has existed in a component since the component's implementation or the last unsuccessful attempt to fix it. While logical and widely used, it is imperfect, as some vulnerabilities may have existed for varying lengths of time. Different estimation approaches may yield different results, thus threatening internal validity.

4.7.1.3 Inclusion of 'Candidate' Code Samples in Training Data

We included 'candidate' code samples in the training data because of data-related challenges, as explained in Subsubsection 4.4.2.3. SARD had not yet confirmed these samples as vulnerable at the time of our analysis. Although this increases the number of vulnerable samples, it poses a risk to internal validity, as some 'candidate' samples may not be genuinely vulnerable. However, SARD does not explicitly discourage the use of ‘candidate’ samples, and we consider the risk posed by false positives to be minimal.

4.7.1.4 Code Churn and Method Signatures

Our code churn detection approach uses method signatures to identify counterparts across releases. Any refactoring that changes a method’s signature in a given release will mean our approach no longer considers the refactored method as a counterpart, treating it as a different method and assessing it accordingly.

4.7.1.5 Shingle Size

We extracted token representations using a shingle size of ‘5’, as preliminary experiments indicated that this size yielded the best results in terms of balancing performance and computational efficiency. Shingle size, a hyperparameter, determines the number of tokens in a shingle. Changing the shingle size will result in different token representations, which may lead to varying analysis outcomes.

4.7.2 External Validity

External validity concerns the generalisability of the study’s findings to other contexts.

4.7.2.1 Generalisation to other Programming Languages

We experimented with a Java-based dataset, using JavaParser to extract token representations. While Java is a popular programming language and JavaParser a widely used tool, generalising our approach to other programming languages may yield different results due to syntactical and semantic differences.

4.7.2.2 Generalisability to Other Software Systems

We focused on the Apache Tomcat 7 software system, a widely used web server. Generalising our approach to other software systems may yield different results due to differences in software size, complexity, and data heterogeneity across systems. Additionally, our vulnerability dataset may not accurately represent all software systems, which could affect its generalisability.

4.7.2.3 Generalisability to Other Vulnerability Datasets

We used the NIST Software Assurance Reference Dataset for our analysis. While SARD is widely recognised, applying our approach to other vulnerability datasets might produce different results.

4.7.2.4 Generalisability to Other Machine Learning Classifiers

We experimented with ten machine learning classifiers, identifying the Random Forest classifier as the best performer. Applying our approach to other classifiers might yield different results, as each classifier has distinct strengths and weaknesses.

4.8 Answer to Research Question 1

Research Question 1: How well does the information retrieval-driven software vulnerability prediction technique perform on a single, multi-release software system dataset for token-based source code representations?

Software industry professionals are well aware of the risks posed by software vulnerabilities and the importance of effective mitigation strategies. Leveraging machine learning to predict vulnerability locations within software presents a cutting-edge solution. This approach enhances security testing efficiency by enabling testers to focus on the most vulnerable components. Despite its promise, AI-driven vulnerability prediction remains limited outside research due to technical constraints and data-related challenges. Nevertheless, concerted efforts from software professionals interested in vulnerability prediction research could revolutionise software security and safeguard our digital environment.

To contribute to advancing AI-driven vulnerability prediction, this chapter proposes an information retrieval technique that leverages token-based software metrics to predict method-level vulnerabilities in software systems. We conducted an empirical study on the Apache Tomcat 7 software system, comprising 76 releases and over 20,000 vulnerable code samples from the NIST Software Assurance Reference Dataset. We extracted token representations and shingles of the methods in the software system and the vulnerable code samples. We then calculated custom software metrics and used them to train and evaluate ten machine learning classifiers.

The results showed that the Random Forest classifier achieved the best predictive performance, with a precision of 0.73, a recall of 0.60, and an F1 score of 0.66 after tuning its hyperparameters.

These findings indicate that the Random Forest classifier is an effective tool for vulnerability prediction, which is in agreement with several other studies [Walden et al., 2014, Scandariato et al., 2014, Kalouptsoglou et al., 2022, Amasaki et al., 2023, Al Debeyan et al., 2022].

Additionally, most metrics in our identified best metrics combination, NDVT, NHS, NTT, NUSM, NVT, SHVSR, and SMR in Tables 4.1 and 4.4, encode vulnerable code patterns, facilitated by our information retrieval-driven technique. This suggests that creating a solution that encodes vulnerable code patterns is crucial to effective vulnerability prediction. This observation explains why supervised machine learning techniques, which utilise knowledge of vulnerable code patterns from training data, remain predominant in vulnerability prediction research, in contrast to unsupervised machine learning techniques, which do not leverage such knowledge.

This study highlights the feasibility of repurposing information retrieval-based techniques for practical vulnerability prediction analysis. These techniques contribute to the development of security-specific metrics and offer insights into alternative metrics that capture various aspects of software quality, including code size and complexity. We anticipate that more vulnerability researchers will adopt these information retrieval-based techniques to enhance actionable vulnerability prediction performance.

Future directions include augmenting training data with a larger vulnerability dataset to allow classifiers to learn a broader range of vulnerable code patterns. However, data quality and quantity-related challenges in software vulnerability research remain significant obstacles, making this strategy challenging. An extension of the work in this chapter will focus on localising the vulnerability dataset to the specific locations of the vulnerability in each method rather than the entire method. The goal is to enhance the performance of the information retrieval-driven vulnerability prediction technique by providing the classifiers with more granular and targeted information about the vulnerable code patterns they need to learn.

We conclude the chapter by explicitly answering the first research question of this thesis, stating that the information retrieval-driven software vulnerability prediction technique performs well on a single, multi-release software system dataset for source code token representation, with the Random Forest classifier achieving the best predictive performance.

Chapter 5

Abstract Syntax Tree (AST)-Based Vulnerability Prediction

This chapter introduces our novel AST-based metrics, specifically Code2Vec-based metrics, for information retrieval-driven vulnerability prediction and assesses their effectiveness. It replicates the vulnerability prediction experiment from Chapter 4 using the Code2Vec technique, comparing its effectiveness with that of the token-based approach. This chapter addresses the second research question of the thesis.

5.1 Introduction

In the Background chapter, we introduced source code representations, discussing their various software engineering applications in areas such as source code classification, code clone detection, bug prediction, and code summarisation. We discussed the source code representation approaches used in our research, including token-based and AST-based representations.

Token-based representations involve tokenising the source code into a sequence of tokens, each representing a specific syntactic element, such as keywords, identifiers, literals, and operators. We explained that due to their simplicity and ease of generation, these representations are widely used in tasks such as code clone detection [Li et al., 2017], bug prediction [Choudhary and Singh, 2017], and code summarisation [Fowkes et al., 2017]. We introduced N-grams as a token-based representation technique involving sequences of N tokens from the source code. We added that character-based N-grams capture morphological information, while word-based N-grams capture semantic information. We also discussed shingling, which involves sequences of N tokens to capture the local context of tokens in the source code.

Following token-based representations, we introduced AST-based representations and explained that they are tree-like structures that represent a program's structure. They represent code elements, such as statements, expressions, and declarations, as nodes, with their relationships represented as edges. ASTs are utilised in program analysis tasks, such as type checking, code generation, and refactoring, as well as in machine learning-based tasks, including code completion and recommendation [Miller, 1995, Jiang et al., 2021, Sommerlad et al., 2008, Liu et al., 2022a].

We also briefly mentioned other source code representation approaches, such as Control Flow Graphs (CFGs) [Zhao et al., 2022, Anju et al., 2010] and Program Dependence Graphs (PDGs) [Czech et al., 2017, Horwitz and Reps, 1992], noting that each approach has its strengths and limitations.

Finally, we explained that machine learning and deep learning tasks benefit significantly from these representations, as algorithms typically require numerical inputs. Therefore, representing source code effectively for feature engineering is crucial for converting textual source code into a format suitable for these algorithms [Hancock and Khoshgoftaar, 2020]. Moreover, the choice of representation depends on the specific software engineering task and the desired level of abstraction because, as pointed out earlier, different representations capture various aspects of the source code, each with its advantages and disadvantages [Samoaa et al., 2022].

In this chapter, we will explore AST representations, a technique renowned for effectively capturing the syntax and semantics of source code. Our goal with this technique is to leverage the advantages of AST-based representations to achieve more accurate vulnerability prediction results.

5.1.1 Chapter Motivation

In the previous chapter, we used token-based representations for our vulnerability prediction task. While our results were promising, with a hyperparameter-tuned precision of 0.73, recall of 0.60, and F1 score of 0.66, we identified limitations in the token-based approach, particularly in capturing the semantic information in the source code. We also noted its limited ability to capture the hierarchical structure and relationships within the code [Panichella et al., 2013, Liu et al., 2022a].

Liu et al. [2022a] emphasised that token-based representations fall short in capturing the syntax and structure of code effectively, unlike AST-based representations, which can capture the hierarchical structure and relationships within the source code. Token-based representations, though more straightforward and language-agnostic, may lack the structural and contextual richness needed for complex analysis. On the other hand, AST-based representations provide a more detailed and context-aware view of source code, enabling complex operations and deeper analysis, albeit with increased complexity and resource requirements. These advantages contribute to the popularity of AST-based representations in software engineering research, as highlighted by Samoaa et al. [2022].

The choice between token-based and AST-based representations is crucial, as it significantly impacts the performance of software engineering tasks. To mitigate the limitations of token-based representations, we suggested augmenting tokenisation with techniques like shingling. However, this augmentation may not fully capture the semantic information in the source code. Besides, determining the optimal shingle size can be challenging and may vary depending on the software system. Even more, computational and storage costs must be considered when generating and storing shingles of varying sizes. Given these limitations, this chapter explores the Code2Vec representation technique, an AST-based approach introduced by Alon et al. [2019].

5.1.2 Research Question

In this chapter, we will reproduce the vulnerability prediction experiment from Chapter 4 using the Code2Vec representation technique. Instead of the token-based approach, we will use Code2Vec to represent the source code of the target software system and the vulnerability dataset. While the overall methodology remains the same, this change in representation necessitates adjustments in tools, data preprocessing, feature engineering, and information retrieval setup. These adjustments will be highlighted as we proceed. The goal remains to predict software vulnerabilities using machine learning algorithms. To this end, we will address the second research question of this thesis:

> *How well does the information retrieval-driven software vulnerability prediction technique perform on a single, multi-release software system dataset for Abstract Syntax Tree-based source code representations?*

This research question is similar to the first one, which focuses on token-based representation. The difference lies in the representation technique used. The results will provide a comparative analysis of the effectiveness of Code2Vec representations in predicting software vulnerabilities compared to the token-based approach and offer insights into the adaptability of our information retrieval-driven vulnerability prediction technique to different source code representations.

5.1.3 Research Scope

This chapter covers the following scope:

- **Programming Language:** The datasets used are exclusively written in Java.
- **Method-Level Vulnerability Prediction:** This study focuses on predicting method-level vulnerabilities within Java methods. Vulnerabilities from other sources, such as web services, annotations, and configuration files, are not taken into consideration.
- **Within-Project Vulnerability Prediction:** The aim is to predict vulnerabilities within a single software system across multiple releases.
- **Binary Classification:** The study uses binary classification with machine learning techniques to predict whether a method is vulnerable; multi-class classification for specific vulnerability types is not considered.

5.1.4 Significance and Contributions

This chapter advances secure software engineering methodologies by evaluating the effectiveness of the Code2Vec representation technique in predicting software vulnerabilities.

5.1.4.1 Significance of the Study

This study compares the Code2Vec representation technique with the token-based approach for vulnerability prediction, using identical parameters and setup, to provide valuable insights into the effectiveness of Code2Vec and the adaptability of our technique to different source code representations.

5.1.4.2 Contributions

This study introduces the AST-based version of the novel security-relevant vulnerability prediction metrics discussed in Section 4.3. It utilises these metrics in conjunction with an alternative representation technique and evaluates their effectiveness in predicting software vulnerabilities using our information retrieval-driven approach.

5.1.5 Structure of the Chapter

The rest of this chapter is structured as follows: Section 5.2 provides background information on the Code2Vec representation technique. Section 5.3 presents the novel Code2Vec-based information retrieval-driven software metrics, comparing them with token-based metrics where necessary. Section 5.4 details the experiment methodology. Section 5.5 presents the experiment results. Section 5.6 presents the discussion of the results. Section 5.7 outlines the threats to validity. Finally, Section 5.8 answers the second research question of this thesis.

5.2 Background

In Subsections 2.2.2 and 4.3.2, we discussed that many vulnerability prediction studies have used traditional software product metrics as features in AI-driven vulnerability prediction. These metrics, derived from the source code, quantify the software product's characteristics and are believed to capture syntactic traits of code, which can help predict bugs and vulnerabilities. However, the results from these techniques are often underwhelming and not actionable [Shin and Williams, 2008b,a, Morrison et al., 2015, Munaiah et al., 2017, Sultana and Chong, 2019, Al Debeyan et al., 2022, Zimmermann et al., 2010, Shin et al., 2010, Doyle and Walden, 2011, Shin and Williams, 2013, Moshtari et al., 2013, Meneely et al., 2013, Walden et al., 2014, Perl et al., 2015, Younis et al., 2016, Sultana, 2017b, Sultana et al., 2018b, Chong et al., 2019b, Kalouptsoglou et al., 2020]. We noted that the main criticism is that these metrics are designed to quantify the syntactic characteristics of software products rather than specifically for vulnerability prediction. Thus, they fail to capture code semantics, which are crucial for predicting vulnerabilities. As a result, many researchers have emphasised the importance of using code representation techniques that capture both syntax and semantics for more actionable results [Lin et al., 2020a, Zhang et al., 2020]. This has led to a shift from traditional software metrics to alternative methods of source code representation that better capture code semantics for vulnerability prediction. For these reasons and the comparative purposes stated in Chapter Motivation, we explore the Code2Vec representation technique in this chapter.

5.2.1 Code2Vec Representation: A Revisit

In the Background chapter, we introduced the Code2Vec representation technique, which was developed by Alon et al. [2019]. Code2Vec utilises the structured nature of source code, precisely the finite set of node types and tokens, to represent code as paths within abstract syntax trees. Since its introduction, Code2Vec has gained significant traction in software engineering research, particularly for predicting method names based on the method's body.

The key idea behind Code2Vec is to encode a code snippet into a fixed-length vector that captures its semantic properties. This is achieved by decomposing the code into a set of paths within its AST. Code2Vec introduces ‘path contexts,’ and in this chapter, we employed astminer to extract path contexts from the source code of our datasets.

5.2.2 astminer

astminer[1] is an open-source toolkit from JetBrains Research[2] for mining ASTs and analysing software. It provides APIs for extracting, querying, and manipulating ASTs and supports various programming languages, including Java, Kotlin, C/C++, Python, JavaScript, and PHP. We used astminer to extract code representation data for our analysis.

ASTs are tree-like data structures that represent the source code structure and syntax of programming languages. The astminer toolkit parses source code into ASTs and extracts features such as method and variable names, type information, and control flow structures. Researchers can use these features for code comprehension, bug detection, and software refactoring.

astminer implements a Code2Vec output format designed to present extracted data numerically as path contexts. This format uses an ID system to store data efficiently by reusing IDs for different code components. It leverages the structured nature of source code and the finite number of unique node types and tokens in ASTs to represent code snippets as paths. This approach reduces the memory footprint and avoids data duplication in significant mining tasks [Kovalenko et al., 2019].

The ability to manipulate path context data using information retrieval techniques is crucial for implementing our information retrieval-driven vulnerability prediction technique, as it facilitates the extraction of relevant information to predict vulnerabilities.

In this experiment, we utilise information retrieval techniques to extract relevant information from the path context data generated by astminer, thereby devel-

[1] `https://github.com/JetBrains-Research/astminer`
[2] `https://www.jetbrains.com/research/`

oping metrics (features) for vulnerability prediction. The following section presents the novel Code2Vec-based software metrics for vulnerability prediction.

5.3 Code2Vec-Based Metrics

As discussed in Chapter 4, our information retrieval-driven software metrics are a significant contribution. We have developed sixteen custom software metrics driven by information retrieval. These metrics serve as features in machine learning models to predict the vulnerability of software components. The primary difference between this chapter and the previous one is that it utilises Code2Vec representations instead of the token-based representations used previously. The datasets remain consistent, consisting of a target software system and a vulnerability dataset.

The target software system is the system in which we aim to predict vulnerabilities. Our vulnerability dataset comprises thousands of known software vulnerabilities, which we utilise for their patterns of vulnerable source code.

Like in the previous chapter, we categorise the metrics introduced in this chapter into *hit-independent* and *hit-dependent* metrics. A 'hit' denotes fragments of code in a target software system method that match code fragments in vulnerability dataset methods. Specifically, these hits comprise path contexts in this chapter, unlike the hits from the previous chapter, which comprised shingles generated from tokenised source code.

Thus, within this Code2Vec-focused chapter, *a hit refers to the intersection of the path contexts of an arbitrary method in the target software system and the path contexts of a vulnerable method in the vulnerability dataset*. For instance, if the path contexts of the method depicted in Figure 2.4 intersect with those of a vulnerable method in the vulnerability dataset, it is considered a hit.

As in the previous chapter, hit-independent metrics are calculated based on specific attributes (such as code churn, size, and complexity) discernible from the Code2Vec representation of a method in the target software system. Conversely, hit-dependent metrics are those whose calculation depends on the *hit* concept. The following subsections provide a detailed discussion of hit-independent and hit-dependent metrics.

Table 5.1: Code2Vec-Based Hit-Independent Metrics

Code2Vec-Based Hit-Independent Metric	**Abbr.**
Number of Target Software System Path Contexts	NTP
Number of Distinct Target Software System Path Contexts	NDTP
Path Context-Based Instantaneous Code Churn	PICC
Path Context-Based Relative Instantaneous Code Churn	PRICC
Number of Target Software System Diff Path Contexts	NTDP
Number of Target Software System Distinct Diff Path Contexts	NTDDP
Path Context Relative Uniqueness	PRU

Table 5.2: Code2Vec-Based Hit-Independent Metrics Code Attributes of Concern

Code2Vec-Based Hit-Independent Metric Abbr.	**Attributes of Concern**		
	Churn	*Intricacy*	*Size*
NTP			✓
NDTP		✓	✓
PICC	✓		
PRICC	✓		
NTDP			✓
NTDDP		✓	✓
PRU		✓	

5.3.1 Code2Vec-Based Hit-Independent Metrics

In this chapter, we developed seven hit-independent metrics, detailed in Table 5.1. The source code attributes relevant to each metric are summarised in Table 5.2.

5.3.1.1 Number of Target Software System Path Contexts (NTP)

The NTP for a method in the target software system is the total count of path contexts in its Code2Vec representation. NTP is the most straightforward metric we developed, serving as an indicator of the method's size. For example, the NTP for the method in Figure 2.4 is 7. See Subsubsection 4.3.1.1 for the hypothesis.

For a method m_t in the target software system with a *multiset* of path contexts $P_t(m_t)$, NTP is defined as follows:

$$\text{NTP} = |P_t(m_t)|$$

The NTP metric's hypothesis was inspired by several studies, including Shin et al. [2010], Giger et al. [2012], Morrison et al. [2015], Pascarella et al. [2018], and Du et al. [2019]. However, we conceptualised its information retrieval-based design and implementation.

5.3.1.2 Number of Distinct Target Software System Path Contexts (NDTP)

The NDTP for a method in the target software system is the count of *unique* path contexts in its Code2Vec representation. Unlike NTP, which measures the number of path contexts, NDTP focuses on their diversity. This metric indicates the size and diversity of code elements within a method. More distinct path contexts suggest greater intricacy in the method's implementation. For example, the NDTP of the method in Figure 2.4 is 7, as all path contexts are unique. See Subsubsection 4.3.1.2 for the hypothesis.

For a method m_t in the target software system with a *set* of path contexts $P_t'(m_t)$, NDTP is defined as:

$$\text{NDTP} = |P_t'(m_t)|$$

The NDTP metric was inspired by the NTP metric, which in turn was inspired by several studies, including Shin et al. [2010], Giger et al. [2012], Morrison et al. [2015], Pascarella et al. [2018], and Du et al. [2019]. However, we conceptualised its information retrieval-based design and implementation.

5.3.1.3 Path Context-Based Instantaneous Code Churn (PICC)

A target software system method's PICC is defined as the number of times its Code2Vec representation has changed throughout its history. The code churn metric quantifies how often developers rewrite their code over time, measured by version control system check-ins or the number of lines of code added, deleted, or modified [Shin et al., 2010]. Many studies have used code churn in vulnerability prediction and have suggested that higher churn correlates with higher vulnerability [Zimmermann et al., 2010, Shin et al., 2010, Shin and Williams, 2013, Meneely et al., 2013, Morrison et al., 2015].

Unlike the token-based analysis in the previous chapter, where any code change affects the representation, regardless of how trivial, Code2Vec representations do not change as frequently. Minor syntactic changes, such as making a method ‘static’ or adding/removing an annotation, often do not affect the Code2Vec representation. This aligns with the idea that AST-based representations do not capture every syntactic detail.

Our PICC implementation increments a method’s PICC by one for every release in which its Code2Vec representation changes. Thus, PICC counts the number of times a method’s source code has changed throughout the target software system’s history, as reflected in the Code2Vec representation. See Subsubsection 4.3.1.3 for the hypothesis.

Let m_t represent a method in a target software system, and let $\delta(m_t)$ represent the set of releases in which m_t’s Code2Vec representation changed. The PICC for m_t is evaluated as follows:

$$\mathrm{PICC(m_t)} = |\delta(m_t)|$$

The PICC metric’s hypothesis was inspired by several studies, including Shin et al. [2010], Giger et al. [2012], Morrison et al. [2015], Pascarella et al. [2018], and Du et al. [2019]. However, we conceptualised its information retrieval-based design and implementation.

5.3.1.4 Path Context-Based Relative Instantaneous Code Churn (PRICC)

A target software system method’s PRICC is the ratio of times its Code2Vec representation has changed to the total number of releases in which the method has appeared. PRICC is relative, unlike the PICC metric, which is absolute. This relative characteristic makes it more effective in indicating how frequently a method has evolved compared to others.

Let m_t represent a method in a target software system, and let N represent the total number of releases in which m_t appears. The PRICC for m_t is evaluated as follows:

$$\mathrm{PRICC(m_t)} = \frac{\mathrm{PICC(m_t)}}{N}$$

The TRICC metric in Chapter 4 is the token-based equivalent of PRICC. In that chapter, we discussed how N could threaten the validity of machine learning analysis through data leakage, depending on the aim and context of the analysis. This discussion is also relevant to PRICC, so refer to Subsection 4.6.3 for more details on the implications.

For a general illustration of PRICC, see Subsubsection 4.3.1.4, Figure 4.3, and Table 4.3.

The PRICC metric was inspired by the PICC metric, which in turn was inspired by several studies, including Shin et al. [2010], Giger et al. [2012], Morrison et al. [2015], Pascarella et al. [2018], and Du et al. [2019]. However, we conceptualised its information retrieval-based design and implementation.

5.3.1.5 Number of Target Software System Diff Path Contexts (NTDP)

A target software system method's NTDP is the count of path contexts in the symmetric difference between its current and previous release's path contexts. The NTDP metric measures the magnitude of changes between two consecutive method releases. See Subsubsection 4.3.1.5 for the hypothesis.

Let m_t represent a method in the target software system, and let m_{t-1} represent its previous release.

The NTDP metric for m_t is expressed as follows:

$$\mathrm{NTDP(m_t)} = |P_t(m_t)\,\Delta\,P_{t-1}(m_{t-1})|$$

Here, Δ is the symmetric difference operator that returns a *multiset* of elements present in either of the two sets but not in both. $P_t(m_t)$ and $P_{t-1}(m_{t-1})$ represent the path contexts of m_t and m_{t-1}, respectively.

The NTDP metric's hypothesis was inspired by several studies, including Shin et al. [2010], Giger et al. [2012], Morrison et al. [2015], Pascarella et al. [2018], and Du et al. [2019]. However, we conceptualised its information retrieval-based design and implementation.

5.3.1.6 Number of Target Software System Distinct Diff Path Contexts (NTDDP)

A target software system method's NTDDP is the count of *unique* path contexts in the symmetric difference between its current and previous release's path contexts. Unlike NTDP, which measures the magnitude of changes, NTDDP measures the diversity of code elements involved in those changes.

The NTDDP metric for m_t is defined as follows:

$$\mathrm{NTDDP(m_t)} = |P'_t(m_t)\,\Delta\,P'_{t-1}(m_{t-1})|$$

Here, Δ is the symmetric difference operator that returns a *set* of elements present in either of the two sets but not in both, and $P'_t(m_t)$ and $P'_{t-1}(m_{t-1})$ represent the unique path contexts of m_t and m_{t-1}, respectively.

This metric was inspired by the NTDP metric, which in turn was inspired by several studies, including Shin et al. [2010], Giger et al. [2012], Morrison et al. [2015], Pascarella et al. [2018], and Du et al. [2019]. However, we conceptualised its information retrieval-based design and implementation.

5.3.1.7 Path Context Relative Uniqueness (PRU)

The PRU metric quantifies the distinctiveness of path contexts in a method's Code2Vec representation within a target software system. Inspired by information retrieval techniques, the metric is used for method-level vulnerability prediction analysis based on the Term Frequency-Inverse Document Frequency (TF-IDF) approach. The underlying premise is that developers are more prone to introducing vulnerabilities when using complex and rarely utilised programming language features in the context of a given software system or codebase. The PRU metric captures a method's complexity based on its composition of rare code elements compared to other methods in the software system or codebase.

The TF-IDF measure, commonly used in information retrieval, quantifies a word's significance to a document within a corpus. The TF-IDF hypothesis posits that a word is pertinent to a specific document if it appears frequently in that document but infrequently in other documents within the same corpus.

We applied the TF-IDF technique analogously to method-level vulnerability prediction to develop the PRU metric. This analogy is based on three primary components: word, document, and corpus.

In our context, the target software system is the system in which we aim to predict vulnerabilities. Our target software system dataset comprises methods represented using Code2Vec representations, which consist of path contexts.

In this analogy, a word corresponds to a path context, a document to a method, and a corpus to our target software system dataset. Since the analysis pertains to method-level (not class-level) vulnerability prediction, we disregard the classes. We then employ the TF-IDF technique to derive weightings for each method's path context to ascertain their significance to the method. In this context, 'significance' denotes how unique a path context is to the method compared to other methods.

For example, in a given software system or codebase, the Java keyword 'void' will likely appear in every standard method designed not to return a value. In contrast, the keyword 'synchronized' is more likely to appear in a complex method performing a specific operation with threads. Therefore, in a 'synchronized' void method, the 'synchronized' keyword will have a higher TF-IDF weighting than the word 'void' because more methods in the software system will feature the 'void' keyword than the 'synchronized' keyword. If we calculate the average PRU for both a void method and a 'synchronized' void method, the 'synchronized' void method will have a higher PRU due to its increased complexity.

The PRU signifies a method's uniqueness compared to other methods in a software system. Evaluating this metric is comparable to finding the harmonic mean of all the TF-IDF weightings of the words in a document. Thus, a method's PRU is the harmonic mean of all TF-IDF weightings for the path contexts in its Code2Vec representation. The PRU determines how obscure a method is compared to others. A high PRU value suggests the method features advanced, specialised, and rarely applied programming language concepts compared to other methods in the software system or codebase. See Subsubsection 4.3.1.7 for the hypothesis.

Let p represent a path context in a target software system's method, and let m represent a *multiset* of the method's path contexts. The term frequency, $\mathrm{tf}(p,m)$, is the relative frequency of path context p within method m.

It is expressed as follows:

$$\mathrm{tf}(p,m) = \frac{f_{p,m}}{\sum_{p' \in m} f_{p',m}}$$

Here, $f_{p,m}$ represents the raw count of path context p in method m, and $\sum_{p' \in m} f_{p',m}$ denotes the total number of path contexts in m.

Let M represent a *multiset* of all methods' path contexts in our target software system.

The inverse document frequency, $\mathrm{idf}(p,M)$, is expressed as follows:

$$\mathrm{idf}(p,M) = \log \frac{N}{|m \in M : p \in m|}$$

Here, N is the total number of methods in the target software system, $N = |M|$, and $|m \in M \,:\, p \in m|$ is the total number of methods that feature p.

The TF-IDF for a path context, $\mathrm{tfidf}(p,m,M)$, is calculated as follows:

$$\mathrm{tfidf}(p,m,M) = \mathrm{tf}(p,m) \cdot \mathrm{idf}(p,M)$$

Finally, a method's PRU, $\mathrm{PRU}(m,M)$, is obtained by calculating the harmonic mean of the TF-IDF values of all its path contexts:

$$\mathrm{PRU}(m,M) = \frac{f_{p,m}}{\sum_{p' \in m} \frac{1}{\mathrm{tfidf}(p',m,M)}}$$

As before, $f_{p,m}$ is the raw count of a path context in a method's representation. $\sum_{p' \in m} \frac{1}{\mathrm{tfidf}(p',m,M)}$ is the sum of the inverses of the TF-IDF values for all p in m.

We exclusively developed the PRU metric's conceptualisation, software vulnerability prediction contextualisation, hypothesis, design, and implementation, drawing inspiration from how the TF-IDF technique tends to assign higher weightings to rare words in information retrieval.

Table 5.3: Code2Vec-Based Hit-Dependent Metrics

Code2Vec-Based Hit-Dependent (Security-Relevant) Metric	Abbr.
Number of Hit Path Contexts	NHP
Number of Distinct Hit Path Contexts	NDHP
Number of Vulnerability Dataset Path Contexts	NVP
Number of Distinct Vulnerability Dataset Path Contexts	NDVP
Target Software System Method-to-↪ Vulnerable Dataset Method Path Context Similarity Ratio	TVPSR
Path Context Hits-to-Target Software System Method Similarity Ratio	PHTSR
Path Context Hits-to-Vulnerable Dataset Method Similarity Ratio	PHVSR
Number of Path Context Matches	NUPM
Path Context Match Ratio	PMR

Table 5.4: Code2Vec-Based Hit-Dependent Metrics Code Attributes of Concern

Code2Vec-Based Hit-Dependent Metric Abbr.	Attributes of Concern		
	Intricacy	*Similarity*	*Size*
NHP		✓	✓
NDHP	✓	✓	✓
NVP		✓	✓
NDVP	✓	✓	✓
TVPSR		✓	
PHTSR		✓	
PHVSR		✓	
NUPM		✓	
PMR		✓	

5.3.2 Code2Vec-Based Hit-Dependent Metrics

As in the previous chapter, we developed nine security-aware metrics for this chapter to leverage the knowledge of known vulnerabilities in our vulnerability dataset. Previously, we defined a hit as the intersection of the path contexts of a target software system method and those of a vulnerable method in a vulnerability dataset. Using this knowledge, we developed hit-dependent metrics to measure the similarity between the path contexts of a target software system method and those of vulnerable methods.

Table 5.3 presents the hit-dependent metrics developed for this chapter, while Table 5.4 outlines their attributes of concern. All hit-dependent metrics except

NUPM and PMR perform a similarity comparison between the path contexts of a target software system method and those of a vulnerable method. NUPM and PMR measure the general distribution of similarities between a target software system method and the methods in the vulnerability dataset. These hit-dependent metrics are security-relevant; while they do not directly measure or indicate the presence of vulnerabilities, they encode the patterns found in vulnerable code, similar to their token-based counterparts.

5.3.2.1 Number of Hit Path Contexts (NHP)

A target software system method's NHP is the number of path contexts it shares with its most similar known vulnerable method in a vulnerability dataset. The NHP metric is straightforward: it counts the path contexts that intersect between a target software system method and the most similar vulnerable method. See Subsubsection 4.3.2.1 for the hypothesis.

Consider a target software system method m_t with a *multiset* of path contexts $P_t(m_t)$, and a matching method m_v from a vulnerability dataset with a *multiset* of path contexts $P_v(m_v)$. Let the hits between $P_t(m_t)$ and $P_v(m_v)$ be h, where $h = P_t(m_t) \cap P_v(m_v)$.

The NHP for the target software system method is expressed as follows:

$$\text{NHP} = |h|$$

The critical point is that with the token-based approach, hit constituents are shingles, whereas with the Code2Vec-based approach in this chapter, they are path contexts.

We exclusively conceptualised the NHP metric, contextualised it within software vulnerability prediction and developed its hypothesis, design and implementation.

5.3.2.2 Number of Distinct Hit Path Contexts (NDHP)

A target software system method's NDHP is the number of *unique* path contexts it shares with its most similar known vulnerable method in a vulnerability dataset.

While NHP measures the total number of shared path contexts, NDHP measures the diversity of the code elements in that intersection.

Consider a target software system method m_t with a *set* of path contexts $P'_t(m_t)$, and a matching method m_v from a vulnerability dataset with a *set* of path contexts $P'_v(m_v)$. Let the hits between $P'_t(m_t)$ and $P'_v(m_v)$ be h', where $h' = P'_t(m_t) \cap P'_v(m_v)$. NDHP is then expressed as follows:

$$\text{NDHP} = |h'|$$

We exclusively conceptualised the NDHP metric, contextualised it within software vulnerability prediction and developed its hypothesis, design and implementation.

5.3.2.3 Number of Vulnerability Dataset Path Contexts (NVP)

A target software system method's NVP is the number of path contexts in its most similar known vulnerable method in a vulnerability dataset. The NVP metric is similar to the NTP metric introduced in Subsubsection 5.3.1.1, which measures the number of path contexts in a target software system method. The critical difference is that while NTP measures the number of path contexts in the target software system method, NVP measures the number of path contexts in its most similar known vulnerable method. Additionally, NTP is hit-independent, whereas NVP is hit-dependent because it requires a hit between the target software system method and at least one method in the vulnerability dataset to calculate the metric. See Subsubsection 4.3.2.3 for the hypothesis.

NVP is expressed as follows:

$$\text{NVP} = |P_v(m_v)|$$

We exclusively conceptualised the NVP metric, contextualised it within software vulnerability prediction and developed its hypothesis, design and implementation.

5.3.2.4 Number of Distinct Vulnerability Dataset Path Contexts (NDVP)

A target software system method's NDVP is the number of *unique* path contexts in its most similar known vulnerable method within a vulnerability dataset. The NDVP metric is similar to the NDTP metric. However, while NDTP measures the number of distinct path contexts in a target software system method, NDVP measures the number of distinct path contexts in its most similar known vulnerable method. Additionally, NDTP is hit-independent, whereas NDVP is hit-dependent, as it requires a hit between the target software system method and at least one method in the vulnerability dataset to calculate the metric.

NDVP is expressed as follows:

$$\text{NDVP} = |P'_v(m_v)|$$

The NDVP metric was inspired by the NDTP metric, which in turn was inspired by several studies, including Shin et al. [2010], Giger et al. [2012], Morrison et al. [2015], Pascarella et al. [2018], and Du et al. [2019]. However, we conceptualised its information retrieval-based design and implementation.

5.3.2.5 Target Software System Method-to-Vulnerable Dataset Method Path Context Similarity Ratio (TVPSR)

A target software system method's TVPSR is the Jaccard Similarity between its path contexts and the most similar known vulnerable method in a vulnerability dataset.

The TVPSR metric assesses the extent to which a target software system method shares code elements with a known vulnerable method. It considers the distinct path contexts in both methods relative to the total number of distinct path contexts between them. Although it does not directly account for the order and frequency of path contexts, the path contexts derived from the methods' ASTs inherently encode these attributes, capturing the hierarchical structure and relationships within the source code.

Jaccard Similarity is the ratio of the intersection to the union of two sets. In this context, it represents the ratio of the distinct path contexts shared between m_t and

m_v to the total number of distinct path contexts in both methods. See Subsubsection 4.3.2.5 for the hypothesis.

Consider a target software system method m_t with a *set* of path contexts $P'_t(m_t)$, and a matching method m_v from a vulnerability dataset with a *set* of path contexts $P'_v(m_v)$. Let the hits between $P'_t(m_t)$ and $P'_v(m_v)$ be h', where $h' = P'_t(m_t) \cap P'_v(m_v)$.

TVPSR is expressed as follows:

$$\text{TVPSR} = \frac{|h'|}{|P'_t(m_t) \cup P'_v(m_v)|}$$

Or, more derivatively:

$$\text{TVPSR} = \frac{NDHP}{NDTP + NDVP - NDHP}$$

We exclusively developed the TVPSR metric's software vulnerability prediction contextualisation, hypothesis, design, and implementation, drawing inspiration from the Jaccard Similarity technique used in string metrics.

5.3.2.6 Path Context Hits-to-Target Software System Method Similarity Ratio (PHTSR)

The PHTSR of a target software system method is the ratio of the number of path contexts it shares with its most similar known vulnerable method in a vulnerability dataset to the total number of path contexts in the target software system method. The PHTSR metric measures the extent to which a target software system method shares path contexts with a known vulnerable method, indicating how much of the target software system method consists of code elements also found in the vulnerable method. See Subsubsection 4.3.2.6 for the hypothesis.

Consider a target software system method m_t with a *set* of path contexts $P'_t(m_t)$, and a matching method m_v from a vulnerability dataset with a *set* of path contexts $P'_v(m_v)$. Let the hits between $P'_t(m_t)$ and $P'_v(m_v)$ be h', where $h' = P'_t(m_t) \cap P'_v(m_v)$.

PHTSR is expressed as follows:

$$\text{PHTSR} = \frac{|h'|}{|P'_t(m_t)|}$$

Or, more derivatively:

$$\text{PHTSR} = \frac{NDHP}{NDTP}$$

We exclusively conceptualised the PHTSR metric, contextualised it within software vulnerability prediction and developed its hypothesis, design and implementation.

5.3.2.7 Path Context Hits-to-Vulnerable Dataset Method Similarity Ratio (PHVSR)

A target software system method's PHVSR is the ratio of the number of path contexts it shares with its most similar known vulnerable method in a vulnerability dataset to the total number of path contexts in the known vulnerable method. The PHVSR metric measures the extent to which a known vulnerable method comprises the path contexts it shares with a target software system method. While PHTSR evaluates the target software system method, PHVSR focuses on the known vulnerable method.

When evaluating PHTSR, we ask, "To what extent do the unique path contexts in our hits constitute the unique path contexts in our target software system method?" Similarly, when evaluating PHVSR, we ask, "To what extent do the unique path contexts in our hits constitute the unique path contexts in the vulnerable method?" PHVSR determines the extent to which a vulnerable method comprises distinct code elements of a target software system method. See Subsubsection 4.3.2.7 for the hypothesis.

Consider a target software system method m_t with a *set* of path contexts $P'_t(m_t)$, and a matching method m_v from a vulnerability dataset with a *set* of path contexts $P'_v(m_v)$. Let the hits between $P'_t(m_t)$ and $P'_v(m_v)$ be h', where $h' = P'_t(m_t) \cap P'_v(m_v)$.

PHVSR is expressed as follows:

$$\text{PHVSR} = \frac{|h'|}{|P'_v(m_v)|}$$

Or, more derivatively:

$$\text{PHVSR} = \frac{NDHP}{NDVP}$$

We exclusively conceptualised the PHVSR metric, contextualised it within software vulnerability prediction and developed its hypothesis, design and implementation.

5.3.2.8 Number of Path Context Matches (NUPM)

A target software system method's NUPM is the number of known vulnerable methods in a vulnerability dataset that share at least one path context with the method in question. This metric counts the number of vulnerable methods that share one or more path contexts with a given target software system method. See Subsubsection 4.3.2.8 for the hypothesis.

Consider a target software system method m_t with a *set* of path contexts $P'_t(m_t)$ and a vulnerability dataset V containing n known vulnerable methods, $m_{v_1}, m_{v_2}, \ldots, m_{v_n}$, each with *sets* of path contexts $P'_{v_1}(m_{v_1}), P'_{v_2}(m_{v_2}), \ldots, P'_{v_n}(m_{v_n})$.

NUPM is expressed as follows:

$$\text{NUPM} = |\{m_{v_i} \in V \,:\, P'_t(m_t) \cap P'_{v_i}(m_{v_i}) \neq \emptyset\}|$$

We exclusively conceptualised the NUPM metric, contextualised it within software vulnerability prediction and developed its hypothesis, design and implementation.

5.3.2.9 Path Context Match Ratio (PMR)

A method's PMR in the target software system is the ratio of known vulnerable methods in a vulnerability dataset that share at least one path context with the method to the total number of vulnerable methods in the dataset. The PMR metric is similar to the NUPM metric. However, while NUPM focuses on the absolute number of known vulnerable methods that share at least one path context with a target software system method, PMR focuses on the proportion of such methods relative to the total number of methods in the vulnerability dataset. In other words, PMR is NUPM divided by the total number of vulnerable methods in the dataset.

Consider a target software system method m_t with a *set* of path contexts $P'_t(m_t)$ and a vulnerability dataset containing n known vulnerable methods, $m_{v_1}, m_{v_2}, \ldots, m_{v_n}$.

PMR is expressed as follows:

$$\text{PMR} = \frac{|\{m_{v_i} \in V : P'_t(m_t) \cap P'_{v_i}(m_{v_i}) \neq \emptyset\}|}{n}$$

Or, more derivatively:

$$\text{PMR} = \frac{NUPM}{n}$$

We exclusively conceptualised the PMR metric, contextualised it within software vulnerability prediction and developed its hypothesis, design and implementation.

In the following subsection, we will illustrate how we calculate the hit-dependent metrics using an example.

5.3.3 Code2Vec-Based Metrics Calculation: An Illustration

In Subsection 4.3.3, we demonstrated how to calculate the token-based metrics using one of the most well-known software vulnerabilities: SQL Injection. In that chapter, we used two code snippets to illustrate the calculation of the token-based metrics: one representing a hypothetical target software system method and the other, a hypothetical known vulnerable method from a vulnerability dataset. We excluded specific metrics from the example that required more extensive target software system information than what is representable within a method body, such as TICC, TRICC, NTDT, NTDDT, TRU, NUSM, and SMR. We pointed out that metrics such as TICC, TRICC, NTDT, and NTDDT require information on an entire method's change history, and calculating the TRU metric necessitates information on the entire target software system. Thus, we could not calculate these metrics using only the information represented in the method bodies.

In this chapter, we will utilise the same hypothetical target software system method and known vulnerable method (see Listings 4.2 and 4.3) to demonstrate the calculation of Code2Vec-based metrics. However, unlike the previous chapter, we will use the Code2Vec representations of the methods, as shown in Figures 5.1 and 5.2. We have highlighted the shared path contexts in bold in these representations.

For simplicity and brevity, we will also exclude metrics that require more extensive information about the target software system than can be represented within

```
3,2,4 1,3,3 1,4,4 5,2,6 1,3,5 1,4,6 7,2,8 1,3,7 1,4,8
   ↪ 32,58,33 32,59,32 32,60,33 32,3,32 25,45,26
   ↪ 25,61,34 26,38,34 32,8,35 2,11,32 2,12,35 1,3,2
   ↪ 16,15,17 16,16,2 16,17,10 17,18,2 17,19,10 2,20,10
   ↪ 36,21,16 36,22,17 36,23,2 36,21,10 1,3,36 3,7,5
   ↪ 3,7,7 5,7,7 11,8,12 11,9,3 11,9,5 11,9,7 12,10,3
   ↪ 12,10,5 12,10,7 13,11,11 13,12,12 13,13,3 13,13,5
   ↪ 13,13,7 13,3,13 13,8,14 15,11,13 15,12,14 15,3,15
   ↪ 15,8,19 15,9,36 19,10,36 9,11,15 9,12,19 9,13,36
   ↪ 9,3,9 9,31,21 9,32,10 21,33,10 9,34,22 21,35,22
   ↪ 10,36,22 9,8,23 9,8,24 9,37,2 24,38,2 2,15,17
   ↪ 2,39,9 2,40,24 2,41,2 17,42,9 17,43,24 17,44,2
   ↪ 25,45,26 26,46,2 26,47,17 13,8,37 15,8,37 9,8,37
   ↪ 32,8,37 30,62,38 30,49,39 30,48,27 30,63,22
   ↪ 38,64,39 38,65,27 38,66,22 39,67,27 39,68,22
```

Figure 5.1: Code2Vec Representation of the Method in Listing 4.2

```
1,1,2 3,2,4 1,3,3 1,4,4 5,2,6 1,3,5 1,4,6 7,2,8 1,3,7
   ↪ 1,4,8 9,5,10 9,3,9 9,6,10 3,7,5 3,7,7 5,7,7 11,8,12
   ↪  11,9,3 11,9,5 11,9,7 12,10,3 12,10,5 12,10,7
   ↪ 13,11,11 13,12,12 13,13,3 13,13,5 13,13,7 13,14,13
   ↪ 13,8,14 15,11,13 15,12,14 15,14,15 16,15,17 16,16,2
   ↪  16,17,10 17,18,2 17,19,10 2,20,10 18,21,16
   ↪ 18,22,17 18,23,2 18,21,10 1,3,18 15,8,19 15,9,18
   ↪ 19,10,18 9,24,20 9,25,15 9,26,19 9,27,18 20,28,15
   ↪ 20,29,19 20,30,18 9,31,21 9,32,10 21,33,10 9,34,22
   ↪ 21,35,22 10,36,22 9,8,23 9,8,24 9,37,2 24,38,2
   ↪ 2,15,17 2,39,9 2,40,24 2,41,2 17,42,9 17,43,24
   ↪ 17,44,2 25,45,26 26,46,2 26,47,17 27,1,28 28,8,29
   ↪ 30,48,9 30,49,31 30,50,1 30,51,2 30,52,9 9,53,31
   ↪ 9,54,2 31,55,1 31,56,2 31,57,9
```

Figure 5.2: Code2Vec Representation of the Method in Listing 4.3

a method body. These excluded metrics are the Code2Vec counterparts of the token-based metrics we excluded in the previous chapter: PRICC, NTDP, NTDDP, PRU, NUPM, and PMR.

Table 5.5 presents the calculated Code2Vec-based metrics for the hypothetical target and the known vulnerable methods. The full names of the metrics are listed in Table 5.1 and Table 5.3 for reference.

The NTP metric, representing the number of path contexts in the target software system method, is 91. This value is obtained by counting the path contexts in the Code2Vec representation of the target software system method, as shown in Figure 5.1.

Table 5.5: Example Code2Vec-Based Metrics Calculation

Hit-Independent Metrics	**Metric Value**
NTP	91
NDTP	90
Hit-Dependent Metrics	**Metric Value**
NHP	47
NDHP	47
NVP	86
NDVP	86
TVPSR	$\frac{47}{129}$
PHTSR	$\frac{47}{90}$
PHVSR	$\frac{47}{86}$

The NDTP metric, indicating the number of distinct path contexts in the target software system method, is 90. This value is derived by counting the distinct path contexts in the Code2Vec representation of the target software system method.

The NHP, a metric concerned with counting the number of shared path contexts between the target software system method and the known vulnerable method, is 47. This value is obtained by counting the intersecting path contexts between the Code2Vec representations of the target and vulnerable methods, as shown in Figures 5.1 and 5.2. The matching path contexts are highlighted in bold in the representations.

The NDHP metric, representing the number of distinct shared path contexts between the target and known vulnerable methods, is also 47.

The NVP metric, indicating the number of path contexts in the known vulnerable method, is 86. This value is derived by counting the path contexts in the Code2Vec representation of the vulnerable method.

The NDVP metric, representing the number of distinct path contexts in the known vulnerable method, is also 86.

The TVPSR metric, which is the target software system method-to-vulnerable method path context similarity ratio, is calculated as $\frac{47}{(90+86)-47}$, which simplifies to $\frac{47}{129}$. 90 and 86 are the number of distinct path contexts in the target software system and known vulnerable methods, respectively.

The PHTSR metric, representing the ratio of shared path contexts to the number of path contexts in the target software system method, is $\frac{47}{90}$.

The PHVSR metric, representing the ratio of shared path contexts to the number of path contexts in the known vulnerable method, is $\frac{47}{86}$.

This example illustrates how we calculate the Code2Vec-based metrics. In the following subsection, we will provide a brief comparison of the token-based and Code2Vec-based metrics to highlight the key differences between the two approaches.

5.3.4 Token-Based versus Code2Vec-Based Approaches

Chapter 4 introduced token-based and shingle-based metrics using token-based representations to predict vulnerabilities in software systems. These metrics capture the structural and evolutionary intricacies of methods in a target system and their similarities to known vulnerable methods in a vulnerability dataset. They are categorised into hit-independent and hit-dependent metrics. Hit-independent metrics are calculated using only the attributes of the target software system methods. In contrast, hit-dependent metrics leverage attributes of the target software system methods and their most relevant methods in the vulnerability dataset.

This chapter introduces AST-based metrics using Code2Vec representations to represent source code. Similar to their token-based counterparts, these metrics capture the structural and evolutionary intricacies of methods in a target system and their similarities to known vulnerable methods in a vulnerability dataset. The AST-, or more specifically, Code2Vec-based metrics, are also divided into hit-independent and hit-dependent categories.

The primary difference between the token-based and Code2Vec-based metrics lies in the method of representation. Token-based metrics use sequences of tokens extracted from the source code and shingles derived from these sequences. In contrast, Code2Vec-based metrics utilise representations generated from the methods' ASTs. Code2Vec representations capture the hierarchical structure and relationships within the source code, providing a potentially more detailed and context-aware method representation.

This subsection provides a comparative summary of the token-based and Code2Vec-based metrics, highlighting their key differences to clarify the distinction between the two approaches.

Table 5.6: Summary of Differences: Token-Based versus Code2Vec-Based Metrics

	Token-Based	**Code2Vec-Based**
Metrics	***Token and Shingle-based***	***Path Context-based***
Query Constituents	Shingles	Path Contexts
Document Index	Shingles	Path Contexts
Hit Constituents	Shingles	Path Contexts
Code Churn Sensitivity	High	Relatively Lower
Token Extraction Tool	JavaParser	Not Applicable
Shingle Generation Tool	Apache Lucene	Not Applicable
Code2Vec Extraction Tool	Not Applicable	astminer

Table 5.6 summarises the key differences between the token-based and Code2Vec-based metrics.

Token-based metrics use tokens and shingles as query constituents, document index, and hit constituents. They are more sensitive to code churn because they rely on token sequences extracted directly from the source code. Even a minor change in the raw source code can significantly affect the token sequences and shingles, which in turn can impact the metrics. These metrics require a token extraction tool, such as JavaParser, to extract the token sequences and a shingle generation tool, like Apache Lucene, to generate the shingles.

In contrast, Code2Vec-based metrics use path contexts for query constituents, document index, and hit constituents. They are less sensitive to code churn as they do not capture every minor detail in the source code. Code2Vec representations are generated directly from the ASTs of methods using astminer.

In summary, token-based metrics utilise token and shingle-based representations, whereas Code2Vec-based metrics employ Code2Vec representations of methods.

Following this comparison, we will proceed to the next section, which outlines the methodology used in this chapter.

5.4 Methodology

This section outlines the methodology used in this chapter. We begin with essential preliminary information, followed by information on our datasets, data preprocessing, information retrieval techniques, and machine learning analysis. Given the similarity between the token-based and Code2Vec-based methodologies, this section focuses on the unique aspects of the Code2Vec-based methodology, referring readers to the relevant sections in Chapter 4 for shared aspects.

5.4.1 Overview of the Methodology

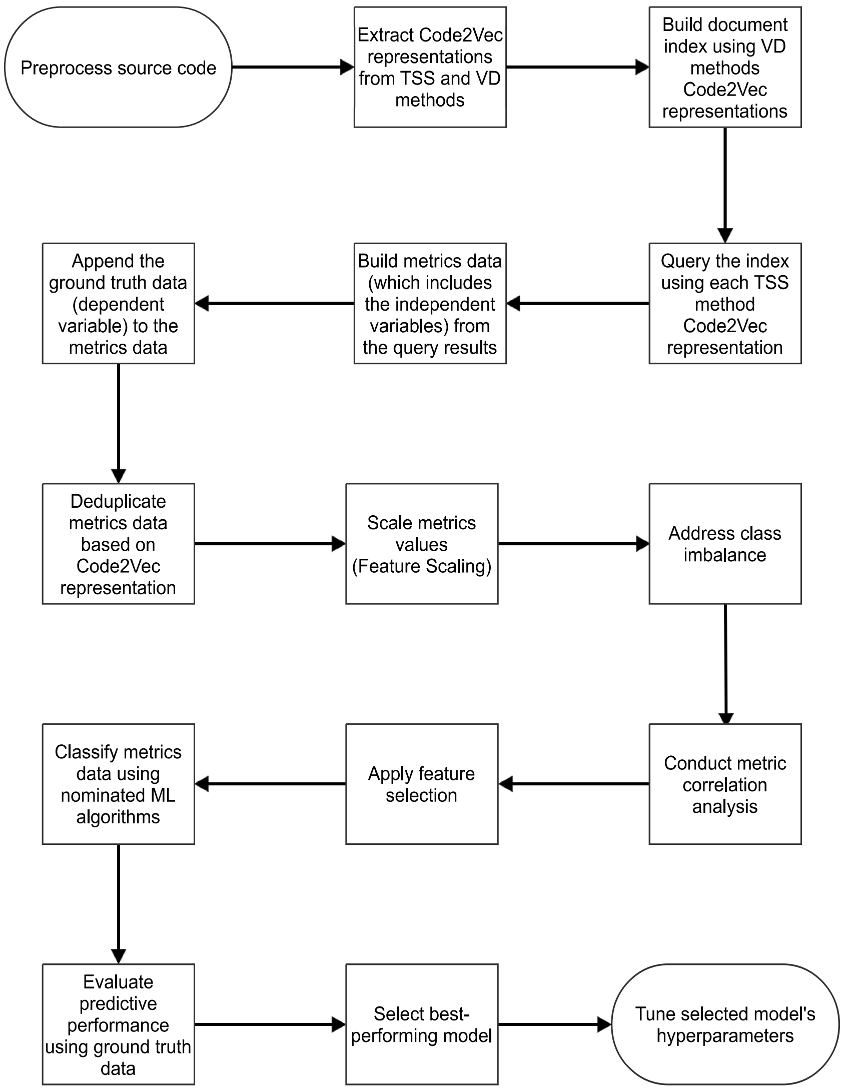

Preprocess source code
Extract Code2Vec representations from TSS and VD methods
Build document index using VD methods Code2Vec representations
Query the index using each TSS method Code2Vec representation
Build metrics data (which includes the independent variables) from the query results
Append the ground truth data (dependent variable) to the metrics data
Deduplicate metrics data based on Code2Vec representation
Scale metrics values (Feature Scaling)
Address class imbalance
Conduct metric correlation analysis
Apply feature selection
Classify metrics data using nominated ML algorithms
Evaluate predictive performance using ground truth data
Select best-performing model
Tune selected model's hyperparameters

Figure 5.3: Code2Vec-Based Vulnerability Prediction Methodology Overview (Within-Project)

Figure 5.3 provides an overview of our approach, divided into five main phases: source code preprocessing, Code2Vec representation extraction, information retrieval, metrics data development, and machine learning analysis.

5.4.1.1 Source Code Preprocessing

This phase is identical to the source code preprocessing phase of the token-based methodology (see Subsubsection 4.4.1.1).

5.4.1.2 Code2Vec Representation Extraction

After preprocessing, we parsed each source code file in our target software system and the vulnerability dataset. We then extracted the Code2Vec representations of the methods in both systems using astminer. For more information on astminer, refer to Subsection 5.2.2.

5.4.1.3 Information Retrieval

This phase involved constructing and querying a document index to apply information retrieval techniques to the path contexts in the Code2Vec representations of the methods in our target software system and vulnerability datasets. The setup is similar to the token-based methodology's information retrieval phase (see Subsubsection 4.4.1.4). The primary difference is that Code2Vec representations are utilised in the construction and querying of the document index.

5.4.1.4 Metrics Data Development

This phase embodies our core contribution: the sixteen metrics we developed. It comprised two subphases: generating metrics data for each method in our target software system and supplementing this data with ground truth information for performance evaluation in the machine learning analysis phase. The phase mirrors that of the token-based methodology (see Subsubsection 4.4.1.5). The only difference is that Code2Vec representations are used instead of token sequences and shingles.

5.4.1.5 Machine Learning Analysis

As in Chapter 4, this phase aimed to:

i Identify the best-performing machine learning classifier for vulnerability prediction.

ii Identify the best-performing combination of metrics for vulnerability prediction.

To achieve these aims, several subphases were completed, including Metrics Data Deduplication, Feature Scaling, Class Imbalance Mitigation, Correlation Analysis, Feature Selection, Classification, Performance Evaluation, Model Selection, and Model Optimisation, similar to the token-based methodology (see Subsubsection 4.4.1.6). The only distinction lies in the metric data deduplication subphase: in the token-based methodology, duplicates were removed based on the token representation of methods, while in the Code2Vec methodology, they were removed based on the Code2Vec representation.

The phases addressed above summarise the methodology employed in this chapter. A comparison with the methodology presented in the previous chapter highlights the differences between the two approaches, as illustrated in Figures 4.6 and 5.3. The following subsections provide detailed information on each step outlined above.

5.4.2 Dataset

This analysis used the same target software system and vulnerability datasets as in Chapter 4. See Subsection 4.4.2 for detailed information on the datasets.

5.4.3 Data Preprocessing

We used astminer to extract Code2Vec representations from the source code in our datasets, subsequently developing information retrieval-based features for machine learning classification.

5.4.3.1 Source Code Path Extraction

The IDs used by the Code2Vec representation elements extracted by astminer, including tokens, paths, and node types, are volatile, meaning that the IDs generated for the same elements differ every time astminer is executed. Therefore, to use the SARD dataset with a target software system for information retrieval-driven vulnerability prediction, it is necessary to ensure that the Code2Vec representation elements from both datasets use IDs generated in the same execution. We designed our setup to extract code representations from the SARD and target software system datasets in a single execution, ensuring consistent IDs across both datasets for the same source code elements.

After extracting source code paths, we applied information retrieval techniques and machine learning classification, which will be discussed later in this section. Before that, we describe the additional steps we took to prepare our data, including data deduplication, data imbalance handling, and ground truth evaluation.

5.4.3.2 Data Deduplication

As in the previous chapter, we deduplicated the Tomcat (training) dataset to prevent data leakage. Refer to Subsubsection 4.4.3.3, Figure 4.3, and Table 4.10 for more details on our deduplication strategy.

To accurately compare the Code2Vec and token-based techniques, we used the exact post-deduplication figures for the target software system dataset, Apache Tomcat 7, as in the previous chapter, including the class distribution. See Table 4.11.

5.4.3.3 Feature Scaling

We applied Min-Max scaling to the metrics data to ensure that all features were on the same scale, as described in Subsubsection 4.4.3.5 in the previous chapter.

5.4.3.4 Software System Vulnerabilities and Data Imbalance

Similar to the previous chapter, we employed the Synthetic Minority Oversampling Technique (SMOTE) to address the class imbalance between vulnerable and non-vulnerable methods in our dataset. Refer to Subsubsection 4.4.3.6 for more information on data imbalance and SMOTE.

5.4.4 Information Retrieval

Following the extraction of Code2Vec representations, as detailed in Subsubsection 5.4.1.2, we applied information retrieval techniques to identify the best-matching methods in the vulnerability dataset for each method in the target software system. The ‘best-matching’ methods share the most path contexts with the target software system methods, indicating a higher likelihood of the same vulnerability. We utilised Apache Lucene, a high-performance, full-featured text search engine library in Java.

5.4.4.1 Document Index Construction

We reused the same number of target software system releases, vulnerability fixes, and indexed vulnerability dataset methods as in the previous chapter to ensure a fair comparison between the Code2Vec and token-based techniques, as shown in Table 4.9. The number of indexed methods in the vulnerability dataset was 20,692, consistent with the previous chapter. The index structure consists of each method’s custom ID mapped to its Code2Vec representation.

5.4.4.2 Apache Lucene Query Construction (BooleanQuery)

Our queries comprised the path contexts in the Code2Vec representation of each method in the target software system. Each query included the path contexts of a method, separated by the ‘OR’ boolean logical operator. For example, the query for the Code2Vec representation shown in Figure 2.4 would be: *"4494,2,5136" OR "2307,1065,154" OR "2307,1066,25" OR "2307,1067,140" OR "154,15,25" OR "154,12,140" OR "25,639,140".*

The ‘OR’ operator in our query strings makes each path context optional, providing flexibility in the information retrieval process. This enables Lucene to retrieve the most relevant SARD methods from the index for any target software system method. The more path contexts that match between a target software system method and a SARD method, the more relevant Lucene considers it, and thus, the higher it ranks.

We then used the results from these queries to calculate the hit-dependent metrics, as described in Subsection 5.3.2.

Table 5.6 summarises the differences between the token-based and Code2Vec-based information retrieval setups.

5.4.5 Machine Learning Analysis

Our machine learning analysis details are identical to those in the previous chapter. To accurately compare the Code2Vec and token-based techniques, we maintained consistency in the machine learning analysis, including the classifiers used, the metrics calculated, and the performance evaluation metrics.

In the machine learning part of the analysis, the representations are irrelevant; the classifiers only use the metrics data extracted from the representations as features.

To avoid redundancy, we will not repeat the details of the machine learning analysis in this chapter. However, as in the previous chapter, we evaluate the classifiers' performance using precision, recall, and F1 scores.

For additional information on the machine learning analysis, refer to Subsection 4.4.5.

5.4.6 Approach to Research Question 2

This chapter addresses our second research question: *How well does the information retrieval-driven software vulnerability prediction technique perform on a single, multi-release software system dataset for Abstract Syntax Tree-based source code representations?*

This question is similar to the first research question from the previous chapter, differing only in that it uses Code2Vec representations instead of token-based ones. Therefore, our approach here is largely similar, with the following objectives:

1. Identify the most suitable classifier for information retrieval-driven, Code2Vec-based method-level vulnerability prediction.

2. Identify the best-performing combination of Code2Vec-based software metrics for vulnerability prediction.

3. Evaluate the impact of hyperparameter tuning on the performance of the best-performing classifier.

Detailed descriptions of the approach will not be repeated here to avoid redundancy. Instead, references to relevant sections of the previous chapter will be provided where necessary.

5.4.6.1 Objective 1

The first objective is to determine the most suitable classifier for information retrieval-driven Code2Vec-based vulnerability prediction at the method level. This aligns with the first objective of the previous chapter, where we utilised precision, recall, and F1 score to evaluate the performance of classifiers, identifying the best-performing classifier based on the highest F1 score (see Subsubsection 4.4.6.1).

5.4.6.2 Objective 2

The second objective is identifying the best-performing combination of Code2Vec-based software metrics for vulnerability prediction. This is similar to the second objective of the previous chapter, where we used Sequential Forward Selection to identify the optimal combination of metrics. Refer to Subsubsection 4.4.6.2 for more details.

5.4.6.3 Objective 3

The third objective is to evaluate the impact of hyperparameter tuning on the performance of the best-performing classifier. This is similar to the third objective of the previous chapter, where we used the Grid Search technique for hyperparameter tuning (see Subsubsection 4.4.6.3).

The following section presents the results of our experiments, organised according to the objectives detailed above. A discussion of the findings, threats to validity, and concluding remarks follow the Results section.

5.5 Results

This section presents the results of our experiments, focusing on the three objectives outlined in Subsection 5.4.6. We discuss the outcomes of the machine learning analysis, focusing on classifier performance, metrics combinations, and the performance of hyperparameter tuning.

5.5.1 Objective 1 Results

Identify the most suitable classifier for information retrieval-driven Code2Vec-based method-level vulnerability prediction.

This objective aimed to identify the best-performing binary classifier based on predictive performance.

5.5.1.1 Evaluation Metrics Trend Analysis

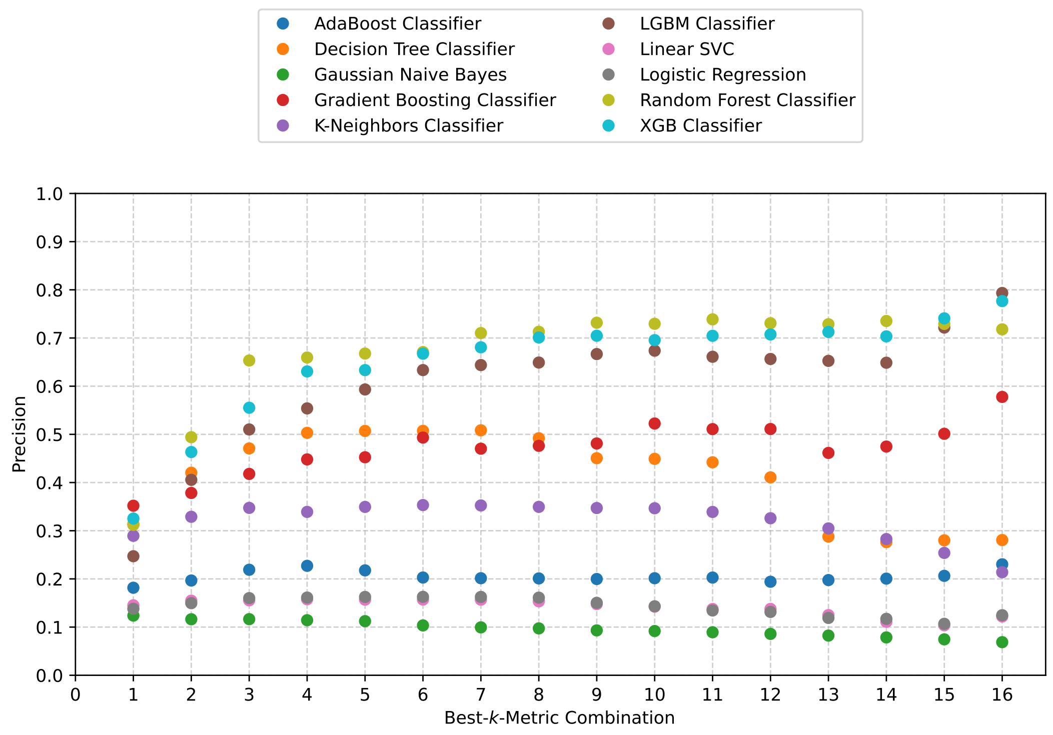


Figure 5.4: Precision Trend across all Best-k-Performing Metrics Combinations

Figure 5.4 shows the precision trends for each classifier's top k metrics combinations. The LGBM classifier achieved the highest precision, nearly 0.80, at $k = 16$, while the Gaussian Naïve Bayes classifier had the lowest precision, around 0.07, at $k = 16$.

The Random Forest classifier consistently performed well across most metrics combinations, slightly outperformed by the Gradient Boosting and XGB classifiers at $k = 1$, $k = 15$ and $k = 16$, and the LGBM classifier at $k = 16$.

The LGBM and XGB classifiers generally showed above-average precision trends across most metrics combinations. The Decision Tree and Gradient Boosting classifiers also exhibited occasional above-average precision, specifically between $k = 4$ and $k = 7$ for the Decision Tree and between $k = 10$ and $k = 12$ for the Gradient Boosting classifier.

The Gaussian Naïve Bayes, Logistic Regression, and Linear Support Vector classifiers performed the worst in precision. The Gaussian Naïve Bayes had the lowest precision across all metrics combinations, while the Logistic Regression and Linear Support Vector classifiers showed similarly poor precision trends.

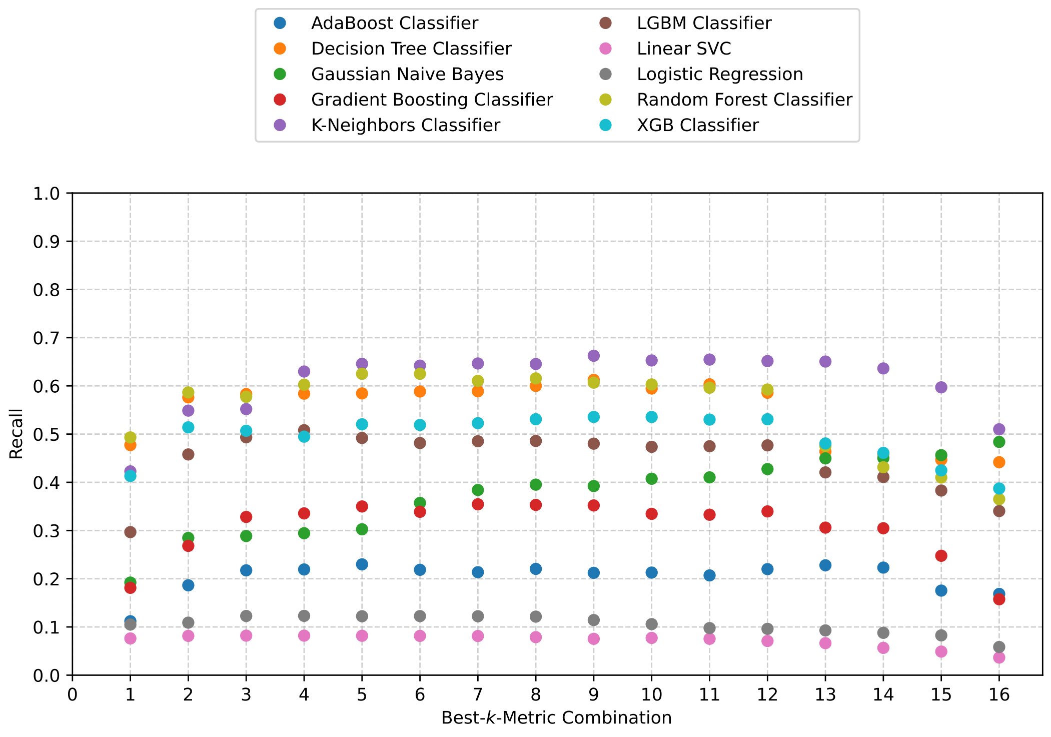


Figure 5.5: Recall Trend across all Best-*k*-Performing Metrics Combinations

Figure 5.5 shows the recall trends for each classifier's top k metrics combinations. The K-Nearest Neighbours classifier achieved the highest recall, approximately 0.66 at $k = 9$, while the Linear Support Vector classifier had the lowest recall, around 0.04 at $k = 16$.

The K-Nearest Neighbours classifier performed well across most metrics combinations, from $k = 4$ to $k = 16$. The Decision Tree and Random Forest classifiers performed better between $k = 1$ and $k = 3$.

The Decision Tree and Random Forest classifiers also had above-average recall trends from $k = 2$ to $k = 12$. The XGB classifier showed average recall trends at $k = 2$ and from $k = 5$ to $k = 12$. Other classifiers mainly exhibited below-average recall trends.

The worst-performing classifiers in terms of recall were the Linear Support Vector and Logistic Regression classifiers. The Linear Support Vector classifier mostly had recall values below 0.1, and Logistic Regression generally had recall values significantly below 0.15.

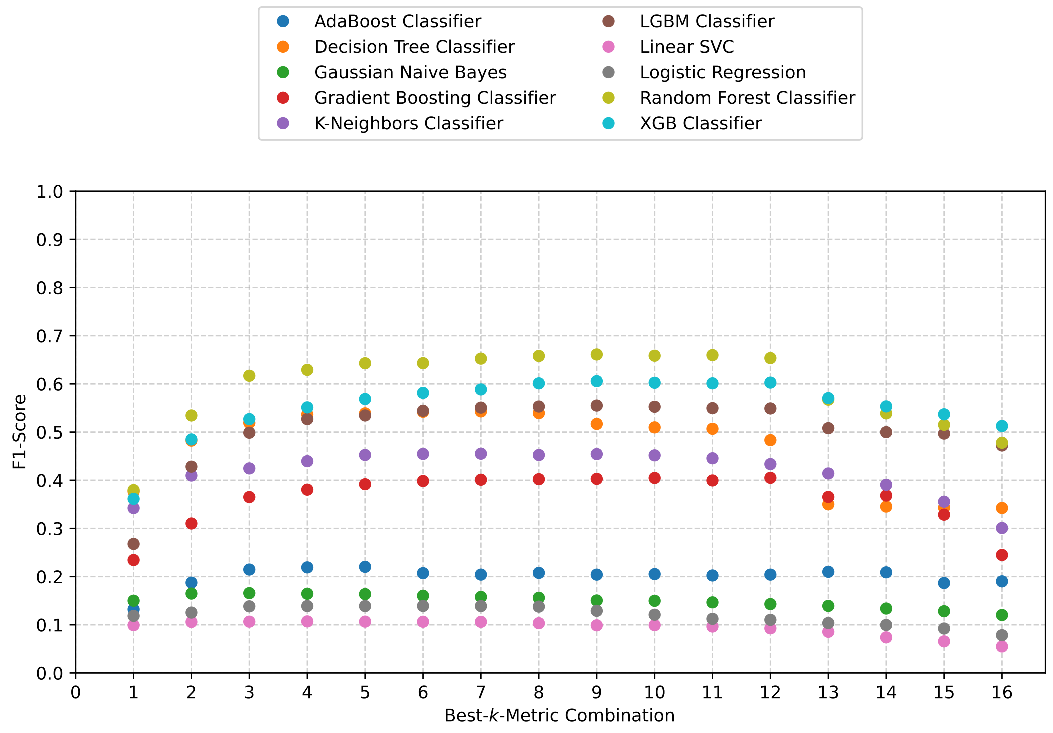


Figure 5.6: F1 score Trend across all Best-k-Performing Metrics Combinations

Figure 5.6 shows the F1 score trends for each classifier's top k metrics combinations. The Random Forest classifier achieved the highest F1 score, approximately 0.66 at $k = 9$, while the Linear Support Vector classifier had the lowest, around 0.05 at $k = 16$.

The Random Forest classifier consistently performed well, especially from $k = 1$ to $k = 12$, but the XGB classifier outperformed it slightly from $k = 13$ to $k = 16$.

The XGB and LGBM classifiers generally achieved above-average F1 scores across most metrics combinations, with the XGB classifier performing particularly well across several k values, second only to the Random Forest classifier. The Decision Tree classifier achieved above-average F1 scores from $k = 3$ to $k = 11$ before declining sharply from $k = 12$ to $k = 16$.

The worst-performing classifiers in terms of F1 score were the Linear Support Vector, Logistic Regression, and Gaussian Naïve Bayes classifiers. The Linear Support Vector classifier had the lowest F1 scores, mostly around 0.1, followed by Logistic Regression and Gaussian Naïve Bayes, with F1 scores generally below 0.2.

Based on the F1 score trends, the Random Forest classifier showed the best predictive performance, achieving the highest F1 score across most top k metrics combinations.

5.5.2 Objective 2 Results

Identify the best-performing Code2Vec-based software metrics combination for vulnerability prediction.

This objective aimed to identify the best-performing combination of software metrics for vulnerability prediction among the sixteen available metrics.

5.5.2.1 Metrics Correlation Analysis

Figure 5.7 displays the correlation matrix of all metrics and the ground truth. The first row and column show the Pearson Correlation Coefficient for the ground truth, while the remaining rows and columns represent the other metrics. The matrix has two levels of grouping: ground truth, hit-independent, and hit-dependent metrics. The hit-independent and hit-dependent metrics are further clustered by their respec-

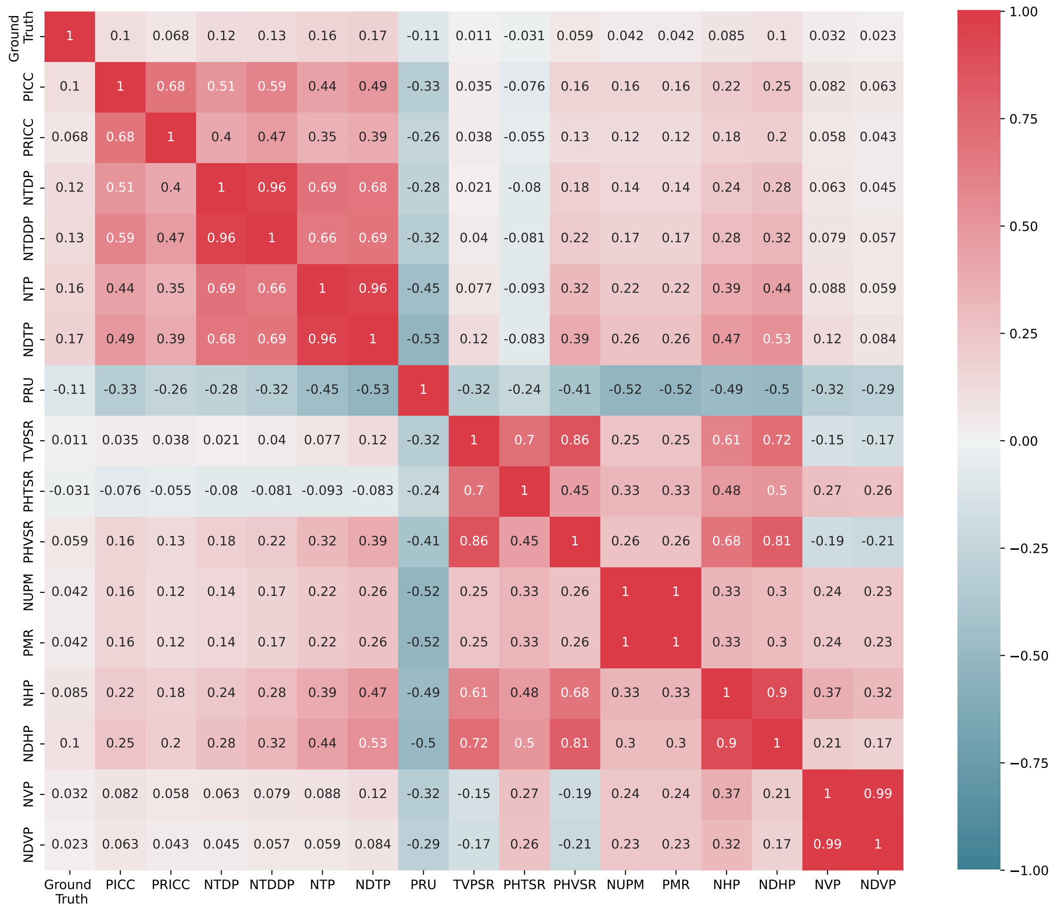


Figure 5.7: Correlation Matrix of all Metrics + Ground Truth

tive constituent metrics. The Pearson Correlation Coefficient measures the linear correlation between two variables, ranging from '-1' (strong negative correlation) to '1' (strong positive correlation), with '0' indicating no correlation. Values in the correlation matrix are categorised as per Table 4.13. The figure includes a scale on the right side, using a diverging colour scheme: red indicates a strong positive correlation, white indicates no correlation, and teal indicates a strong negative correlation.

The main observations from the correlation matrix are presented below. The full names of the metrics are listed in Table 5.1 and Table 5.3.

1. Some cells in Band No. 1 show values close to '1', indicating a strong positive correlation between certain metrics, such as NUPM and PMR, and pairs like

NHP and NDHP. This suggests possible redundancy, which may or may not affect model performance.

2. Negative correlations fall within Band No. 7 and higher, but none are as strong as the positive ones. The highest negative correlation is -0.53 between PRU and NDTP, indicating a moderate negative correlation. PRU measures method uniqueness, while NDTP refers to the number of unique path contexts, which explains the negative correlation.

3. The second-highest negative correlation is -0.52 (PRU and NUPM/PMR), indicating a moderate negative correlation. NUPM counts vulnerable methods sharing a path context, and PMR is their ratio. Methods with many matches to vulnerable methods have lower PRU values.

4. NTP/NDTP and NVP/NDVP are conceptually similar but apply to different datasets. The correlation between NTP and NVP is 0.088, and between NDTP and NDVP is 0.084, both indicating very weak positive correlations. When hits are included, the correlation improves significantly to 0.45 (PHTSR and PHVSR).

5. The TVPSR metric, considering both target system and vulnerability dataset methods, shares a closer bond with PHTSR (0.70) and PHVSR (0.86) than the latter two with each other (0.45).

6. The correlation between NUPM and PMR, i.e., 1 (Band No. 1), is higher than that between PRICC and PICC, 0.68 (Band No. 2). This is because the PMR denominator (total matches in the dataset) is constant, while the PRICC denominator (total system releases) changes, making PRICC values more variable.

7. The ground truth shows no strong correlation with any metrics, with the highest being 0.17 (Band No. 5) with NDTP. Despite weak correlations, classifiers can still perform well, as shown in Figure 5.6, indicating that seemingly redundant metrics may contribute to overall performance.

8. Excluding potentially redundant metrics, such as PMR, NTDDP, NDTP, NDHP, and NDVP, did not necessarily improve classifier performance and sometimes decreased it. Therefore, as noted in the previous chapter, we retained all classification metrics, focusing on feature selection to identify the best-performing combinations.

5.5.2.2 Classifier Performance Analysis

Table 5.7: Best Performance Per Classifier (Sorted by F1 score)

Classifier	**Best k**	**Precision**	**Recall**	**F1 score**
Random Forest classifier	9	0.73171	0.60668	0.66106
XGBoost classifier	9	0.70472	0.53556	0.60564
LightGBM classifier	9	0.66666	0.48016	0.55496
Decision Tree classifier	7	0.50839	0.58884	0.54284
K-Nearest Neighbors classifier	7	0.35246	0.64649	0.45508
Gradient Boosting classifier	12	0.51111	0.33964	0.40499
AdaBoost classifier	5	0.21772	0.22986	0.22031
Gaussian Naive Bayes	3	0.11647	0.28856	0.16568
Logistic Regression	6	0.16279	0.12250	0.13910
Linear Support Vector classifier	4	0.15780	0.08198	0.10700

Table 5.7 presents the best performance of each classifier, sorted by descending F1 score, our primary evaluation metric. The Random Forest classifier achieved the highest F1 score. The XGBoost, LightGBM, and Decision Tree classifiers also performed well, with XGBoost and LightGBM outperforming the Decision Tree. The worst-performing classifiers in terms of F1 score were the Linear SVC, Logistic Regression, and Gaussian Naïve Bayes.

5.5.2.3 Metrics Combination Analysis

Table 5.8 shows the optimal combination of metrics for each classifier, categorised into Hit-Independent and Hit-Dependent sections. A checkmark indicates the best-performing metrics combination for each classifier.

This table reveals that hit-dependent metrics are crucial for most classifiers, frequently appearing in the best-performing combinations. This supports the find-

Table 5.8: Best Metrics Combination Per Classifier

Metric	**Classifiers**									
	AB	***DT***	***GNB***	***GB***	***KN***	***LGBM***	***LSVC***	***LR***	***RF***	***XGB***
Hit-Independent										
NTP			✓	✓	✓	✓	✓	✓	✓	✓
NDTP	✓			✓		✓	✓	✓	✓	
PICC										
PRICC										
NTDP			✓							
NTDDP										
PRU				✓		✓			✓	✓
Hit-Dependent										
NHP		✓		✓	✓	✓				✓
NDHP		✓		✓	✓	✓		✓	✓	✓
NVP	✓	✓		✓	✓	✓			✓	
NDVP	✓	✓	✓	✓	✓				✓	✓
TVPSR				✓			✓	✓		✓
PHTSR				✓			✓			✓
PHVSR	✓	✓		✓		✓			✓	✓
NUPM		✓		✓	✓	✓		✓	✓	✓
PMR	✓	✓		✓	✓	✓		✓	✓	

ings of the previous chapter, which show that hit-dependent metrics outperform hit-independent metrics in terms of predictive performance.

The table also highlights the usefulness of each metric by its frequency in top-performing combinations. For instance, the NTP metric is selected for all classifiers except AdaBoost and Decision Tree, indicating its importance. Conversely, NTDP is only chosen for Gaussian Naïve Bayes, and the code churn metrics, PICC and PRICC, are not selected.

In conclusion, the optimal software metrics combination for vulnerability prediction, as per the second objective, includes those used by the best-performing classifier, the Random Forest classifier. These metrics are NDHP, NDTP, NDVP, NTP, NUPM, NVP, PHVSR, PMR, and PRU.

5.5.3 Objective 3 Results

This objective evaluated the impact of hyperparameter tuning on the performance of the best-performing classifier from the first objective.

5.5.3.1 Parameter Grid and Best Hyperparameter Values

Table 5.9: Parameter Grid and Best Hyperparameter Values

Parameter Grid		
Hyperparameter	***Values***	***Best Value***
bootstrap	True*, False	False
max_depth	None*, 10, 20	20
max_features	‘auto’, ‘sqrt’*	auto
min_samples_leaf	1*, 2, 4	1
min_samples_split	2*, 5, 10	2
n_estimators	100*, 200, 300	200

Table 5.9 shows the parameter grid and best hyperparameter values for the Random Forest classifier. As in the previous chapter, the Random Forest classifier performed well in this Code2Vec-based analysis, so we used the same hyperparameters and values.

The parameter grid includes only six key hyperparameters, as tuning all hyperparameters would be impractical due to time and computational resource constraints. The asterisk (*) denotes the default value in scikit-learn. The last column shows the best-performing values determined by our Grid Search technique (see Subsubsection 4.4.6.3).

We tuned the hyperparameters using the best-performing metrics combination identified in the second objective (NDHP, NDTP, NDVP, NTP, NUPM, NVP, PHVSR, PMR, and PRU) rather than all sixteen metrics.

The parameter grid (first two columns in Table 5.9) in this chapter is identical to the one in the previous chapter (see Table 4.16). However, the ‘max_depth’ hyperparameter has a best value of ‘20’ here compared to ‘None’ in the previous chapter. Also, the ‘n_estimators’ hyperparameter has a best value of ‘200’ here compared to ‘100’ in the previous chapter.

5.5.3.2 Hyperparameter Tuning Results

Table 5.10 shows the pre- and post-hyperparameter tuning results for the Random Forest classifier.

Table 5.10: Pre-and-Post-Hyperparameter Tuning Results for Random Forest classifier

Metric	**Before**	**After**	**Δ%**
Precision	0.73171	0.72111	-1.45
Recall	0.60668	0.62261	2.63
F1 score	0.66106	0.66778	1.02

After tuning, precision decreased by 1.45%, recall increased by 2.63%, and the F1 score increased by 1.02%.

This trend aligns with the previous chapter, where precision decreased by 0.22%, recall increased by 2.23%, and the F1 score increased by 1.42%.

Regarding the third objective, as in the previous chapter, hyperparameter tuning had a mixed impact on the performance of the Random Forest classifier. Precision decreased, recall increased, and the F1 score, our primary evaluation metric, ultimately improved slightly.

5.6 Discussion

This section discusses our results and their implications. We examine the performance of the Code2Vec representation technique, the impact of hit-dependent metrics, the evaluation metrics, and the effect of hyperparameter tuning.

5.6.1 Code2Vec Representation Performance

In this chapter, we explored the effectiveness of the Code2Vec technique for predicting software vulnerabilities. We used an information retrieval-driven approach on a multi-release software system dataset with Code2Vec representations. Our analysis aimed to identify the best classifier for Code2Vec-based vulnerability prediction, determine the optimal combination of metrics, and assess the impact of hyperparameter tuning on the performance of the best classifier.

As in the previous chapter, the Random Forest classifier was the best-performing model. However, it showed slightly better improvements in the F1 score in this chapter compared to the previous one. The token-based approach achieved a maximum post-hyperparameter tuning precision of *0.73472*, recall of *0.59667*, and F1 score of *0.65741*. In contrast, the Code2Vec-based approach achieved a precision of *0.72111*, recall of *0.62261*, and F1 score of *0.66778*.

These results suggest that Code2Vec representations may capture syntactic and semantic information more effectively than token-based approaches, providing a more nuanced understanding of code structure and potential vulnerabilities. This supports Liu et al. [2022a]'s finding about the effectiveness of AST-based representations. However, it is essential to note that the shingle size in the token-based approach may affect the results. In Subsubsection 4.4.3.2, we noted that we employed a shingle size of '5' in the token-based approach, following some experimentation, as it provided the best balance between performance and computational efficiency for our analysis.

Nevertheless, other studies have highlighted the effectiveness of AST-based representations, most notably a critical study by Al Debeyan et al. [2022] that significantly influenced this research. The study tackled the challenge of software vulnerability prediction and detection, using AST information to represent code vulnerabilities for machine learning models. Their goal was to enhance the performance of vulnerability prediction models by using AST N-grams as features, applying binary classification to detect vulnerability status, and multiclass classification to categorise code into different vulnerability types based on Common Weakness Enumeration (CWE) categories. The researchers used a dataset comprising 5,001 real-world vulnerable Java methods from 219 open-source projects, extracting 18 static code metrics and AST N-grams from these methods. They compared the performance of three machine learning models (Random Forest, Naïve Bayes, and SVM) and one deep learning model (DeepBalance) using binary classification, with evaluation metrics including F-measure and MCC. Additionally, they proposed a Random Forest model with multiclass classification to cluster code into different CWE types and evaluated its performance using the same metrics. The study found that the Random Forest model using AST N-grams outperformed the other models in binary classification, achieving an F-measure of 75% and an MCC of 74%. The multiclass classification model using AST N-grams also performed well, with an average F-measure of 61% and an MCC of 63%. The authors concluded that AST N-grams are effective features for improving the predictive performance of vulnerability models and providing more detailed information on vulnerability types. They also suggested further enhancing their approach by incorporating more CWE types, adding code context, and applying transfer learning.

Al Debeyan et al. [2022]'s research is similar to ours in terms of methodology and results. We achieved a comparable F1 score using the same classifier: Random Forest. However, we utilised a different set of novel vulnerability prediction metrics and seven additional machine learning models. Additionally, they considered binary and multiclass classification, while we focused solely on binary classification.

While Al Debeyan et al. [2022]'s study reported impressive results, we observed a data leakage issue in their binary classification analysis, which may have impacted the model's performance. We will address this issue in the following chapter. Nevertheless, their study provides valuable insights into the effectiveness of AST-based representations for vulnerability prediction. These insights support our findings in this chapter and those of other researchers in the field, as seen in the study mentioned earlier by Liu et al. [2022a].

5.6.2 Hit-Dependent Metrics Performance

Another important insight uncovered in this chapter is the significance of hit-dependent metrics, i.e., metrics that use known vulnerabilities to identify patterns indicating security risks. As in the previous chapter, these metrics outperformed hit-independent ones, highlighting the value of incorporating contextual information from known vulnerabilities into a predictive model. Hit-dependent metrics significantly enhanced predictive performance, suggesting that capturing the semantic and syntactic nuances of vulnerable source code is crucial for accurate vulnerability prediction. However, we emphasise that the quality of these metrics depends on the completeness and relevance of the vulnerability dataset from which they are derived. Therefore, comprehensive and up-to-date vulnerability datasets are crucial for accurate vulnerability prediction. This also means that the model must be updated regularly in a real-life scenario to incorporate new vulnerabilities and ensure accurate predictions. Therefore, a CI/CD pipeline that automatically updates the vulnerability dataset and re-trains the model would be beneficial.

5.6.3 A Closer Look at the Evaluation Metrics

Like the previous chapter, the evaluation metrics in this chapter offer a comprehensive view of classifier performance. The Random Forest classifier achieved the highest F1 score, striking the best performance balance between precision and recall in this chapter's Code2Vec-based experiments. This balance is crucial in vulnerability prediction, where false positives and negatives can have severe consequences. The performance of the Random Forest classifier suggests it can effectively identify vulnerabilities while minimising false positives and negatives.

Our observations indicate that the Code2Vec-based representation outperformed the token-based approach in terms of recall and F1 score, subject to the shingle size used in the token-based approach. These improvements are significant since recall and F1 score are crucial metrics in vulnerability prediction. The superior performance of the Code2Vec-based approach in these metrics suggests it can better identify vulnerabilities and minimise false negatives, which is essential in security-critical applications.

However, the precision metric showed a slight decrease. This decrease is not unexpected, as the Code2Vec-based approach may be more sensitive to false positives due to its ability to capture nuanced code semantics. This trend suggests that while Code2Vec enhances the model's ability to identify vulnerabilities (recall), it also increases the number of false positives, leading to a decrease in precision. Nonetheless, the superior recall and F1 score of the Code2Vec-based approach suggest that it may be more effective at identifying vulnerabilities while maintaining reasonable precision. However, as pointed out earlier, shingle sizes in the token-based approach may affect the results, and further investigation is needed to determine the optimal shingle size for vulnerability prediction.

In conclusion, the Code2Vec representation may enhance the model's understanding of complex code structures more effectively, resulting in improved vulnerability detection. This improvement is particularly evident in the recall and F1 score metrics. The increase in recall indicates that the model is more adept at identifying actual vulnerabilities. The improvement in the F1 score highlights the effectiveness of the Code2Vec representation in balancing precision and recall, which is essential for vulnerability prediction. However, the decrease in precision highlights the need for further refinement to reduce false positives.

5.6.4 Hyperparameter Tuning Impact

The Random Forest classifier's hyperparameters remained consistent across chapters, with minor changes in 'max_depth' and 'n_estimators.' This consistency suggests that, despite the differences in metrics data yielded by the Code2Vec and token-based representations, the Random Forest classifier's performance remains stable, highlighting its robustness in vulnerability prediction. We also observed this robustness in the literature, as discussed in Subsection 3.4.3 and supported by the findings in several studies [Walden et al., 2014, Scandariato et al., 2014, Kalouptsoglou et al., 2022, Amasaki et al., 2023, Al Debeyan et al., 2022]. However, we note that different source code representation techniques may require distinct tuning strategies for optimal performance.

Hyperparameter tuning resulted in a slight decrease in precision, an increase in recall, and an overall improvement in the F1 score, consistent with the findings from the previous chapter. This trend indicates that tuning can enhance the Random Forest classifier's performance by balancing precision and recall, improving its vulnerability prediction capabilities.

5.6.5 Implications

Our findings have several implications for vulnerability prediction research and its practical applications. Firstly, the improved performance with Code2Vec suggests that future models should prioritise representations that effectively capture both source code syntax and semantics. This can enhance the model's ability to accurately identify vulnerabilities, which is crucial for security-critical applications. Additionally, the reliance on hit-dependent metrics highlights the need to utilise comprehensive vulnerability datasets to inform the predictive model. Incorporating known vulnerabilities can enhance the model's ability to effectively identify security risks, emphasising the importance of contextual information in vulnerability prediction. Lastly, the changes in hyperparameter optimisation indicate that models using advanced representations, such as Code2Vec, require tailored tuning strategies for optimal results. This highlights the complexity of hyperparameter tuning and suggests that future research should explore strategies tailored to the specific representation used, thereby ensuring optimal performance in vulnerability prediction.

5.6.6 Recommendations

Future research and applications in software vulnerability prediction should adopt Code2Vec or similar AST-based representations that effectively capture syntactic and semantic information. This approach can enhance the model's ability to identify vulnerabilities, thereby improving the security of software systems. Also, combining AST-based representations with other techniques may capture a broader spectrum of code characteristics, potentially improving overall model performance. Future research should explore hybrid representations to leverage the strengths of different techniques, enhancing the model's ability to identify vulnerabilities effectively. Finally, emphasising hit-dependent metrics is crucial for enhancing predictive performance. These metrics significantly boost predictive accuracy by leveraging known vulnerabilities to identify security risks.

5.7 Threats to Validity

This section discusses threats to the validity of our study, focusing on internal and external validity, similar to the previous chapter.

5.7.1 Internal Validity

Internal validity concerns the extent to which observed effects can be attributed to the experimental conditions rather than other factors. Several threats to internal validity were identified.

5.7.1.1 Hyperparameter Tuning

The selection of hyperparameters for the Random Forest classifier might have influenced the outcomes. Although we used Grid Search for tuning, other hyperparameter combinations could yield different results. Future studies should explore a broader range of settings and potentially use automated hyperparameter optimisation techniques.

5.7.1.2 Dataset Characteristics

The specific characteristics of the Apache Tomcat dataset may have impacted the findings. This dataset is from a single software system with multiple releases, which may not be generalisable to other systems. Similar experiments on diverse datasets from different domains are recommended to validate the findings.

5.7.2 External Validity

External validity refers to the extent to which the results can be generalised beyond the specific experimental conditions.

5.7.2.1 Programming Language and Dataset Generalisability

A primary threat is the generalisability to other programming languages and software systems. This study focused exclusively on Java-based systems, specifically the Apache Tomcat dataset. While the results are promising, it remains unclear whether the Code2Vec technique will perform equally well on software in other languages.

5.8 Answer to Research Question 2

Research Question 2: How well does the information retrieval-driven software vulnerability prediction technique perform on a single, multi-release software system dataset for Abstract Syntax Tree-based source code representations?

This chapter examined the effectiveness of the Code2Vec representation technique for predicting software vulnerabilities. We aimed to answer the research question by evaluating the performance of the Code2Vec-based approach on a single, multi-release software system dataset and comparing it with the token-based approach used in the previous chapter.

We preprocessed Apache Tomcat's source code, extracted Code2Vec representations from the methods, and developed software metrics using information retrieval techniques. Various machine learning classifiers were trained using these metrics, and their performance was evaluated based on precision, recall, and F1 score. Hyperparameter tuning was conducted to optimise the best-performing classifier.

The findings indicated that the Code2Vec representation yielded a slight improvement in vulnerability prediction compared to the token-based approach. However, the nominated shingle size in the token-based approach may have influenced the results. The Random Forest classifier emerged as the best-performing model, with precision, recall, and F1 score values of *0.72111*, *0.62261*, and *0.66778*, respectively. Contrastingly, the token-based approach achieved a maximum post-hyperparameter tuning precision of *0.73472*, recall of *0.59667*, and F1 score of *0.65741*. While the token-based approach showed a slightly higher precision, the Code2Vec-based approach achieved better recall and F1 score results, which are critical for security applications. The Code2Vec representation may have captured more detailed code semantics relative to the shingle size of '5' used in the token-based approach, leading to better vulnerability detection.

Our findings also highlighted the importance of hit-dependent metrics in vulnerability prediction, as they significantly enhanced predictive performance compared to hit-independent metrics, as in the previous chapter.

Additionally, the Random Forest classifier's consistent performance across different representations highlights its robustness and suitability for vulnerability prediction tasks.

Finally, our results showed that hyperparameter tuning generally improves predictive performance, as evidenced by the increased F1 score, at the expense of a slight decrease in precision.

Future work should extend the approach to other programming languages, integrate Code2Vec with other representation techniques, explore the impact of different shingle sizes in token-based approaches, conduct longitudinal studies to assess model stability and develop real-time vulnerability prediction tools for practical software development and maintenance applications.

To explicitly answer the second research question of this thesis, the information retrieval-driven software vulnerability prediction technique performed well on a single, multi-release software system dataset for Code2Vec representation, achieving slight but notable improvements in predictive performance, as measured by the F1 score, compared to the token-based approach.

Chapter 6

A Vulnerability Prediction Dataset Generalisability Study

This chapter evaluates the generalisability of our information retrieval-driven vulnerability prediction technique using a dataset of code artefacts from multiple software systems. We assess its performance in a mixed-project setting, with a focus on data quality and quantity limitations. The chapter addresses the third and final research question of this thesis, examining the applicability of the technique across projects and the impact of data-related factors on prediction accuracy.

6.1 Introduction

We provided background on the potential of AI-driven techniques, specifically Supervised Machine Learning, in predicting software vulnerabilities in Section 2.1, stating that these methods utilise historical data from software systems to identify patterns indicative of vulnerabilities and predict future vulnerabilities.

A machine learning-based approach typically involves training a supervised learning model on a dataset of code artefacts labelled with known vulnerabilities. This model can then predict vulnerabilities in other software systems, particularly useful in the early stages of software development [Zhang et al., 2023c].

Despite their largely theoretical effectiveness, predicting vulnerabilities in large-scale software code remains complex, time-consuming, and error-prone, even for domain experts [Zhang et al., 2023b]. A significant challenge lies in obtaining high-quality labelled data. Acquiring extensive datasets of software systems annotated with vulnerabilities is inherently complex, limiting the effectiveness of supervised learning techniques that require large amounts of labelled data to build robust models. Consequently, there are relatively few labelled projects compared to the vast number of unlabelled ones [Nguyen et al., 2024], highlighting the need for alternative approaches to address these data-related challenges. One promising alternative is cross-project prediction.

6.1.1 Cross-Project vs Mixed-Project Vulnerability Prediction

Cross-project or inter-project vulnerability prediction, in its simplest form, involves predicting vulnerabilities in a target software system using a model trained on a *different* source system. This is vital as it allows for predicting vulnerabilities in projects with insufficient data for training a supervised learning model [Malhotra and Meena, 2024]. The model's versatility is also crucial. A model that can be applied across multiple projects (or software systems) is theoretically more adaptable than a within-project prediction model confined to a single project (or software system). Such a generalisable model enhances efficiency by enabling software professionals to use one model for various projects, eliminating the need to train separate models for each project.

In this chapter, we use the term *mixed-project* vulnerability prediction to describe the process of predicting vulnerabilities in a target software system using a model trained on a dataset comprising code artefacts from *multiple* software systems.

We consider the conventional cross-project vulnerability prediction a straightforward subset of mixed-project prediction, where the training data consists of code artefacts from one project. Mixed-project prediction, on the other hand, encompasses a broader range of more complex scenarios where the training data comprises code artefacts from an arbitrary number of projects. Due to the increased data variability in the dataset, mixed-project prediction presents a more significant challenge as it provides a more rigorous test of a model's ability to generalise across diverse software systems. Therefore, we could consider mixed-project prediction a stress test of the model's generalisability.

This chapter examines mixed-project vulnerability prediction and assesses the generalisability of our information retrieval-based vulnerability prediction technique across different software systems using a dataset comprising code artefacts from multiple projects.

6.1.2 Dataset Generalisability: An Introduction

Generalisability refers to the extent to which study findings can be applied to other contexts or settings. In our context, it denotes the applicability of data to various scenarios. Croft et al. [2022] noted that data generalisability measures the external validity of an analysis. This chapter uses the term *data generalisability* to describe how well a vulnerability prediction technique performs across multiple datasets or projects. Specifically, we examine the applicability of our information retrieval-driven vulnerability prediction technique across different projects, exploring its generalisability using a dataset comprising artefacts from various software systems.

6.1.3 Chapter Motivation

In Chapter 4, we introduced an innovative Information Retrieval-driven vulnerability prediction technique, which forms the core of our thesis. This technique utilises information retrieval methods to generate metrics from various source code attributes, including size, complexity, churn, and known vulnerability code patterns. These metrics are then used to predict vulnerabilities. We evaluated this technique using token-based code representations, achieving promising results with a Random Forest classifier: a precision of 0.73, a recall of 0.60 and an F1 score of 0.66. In Chapter 5, we extended the technique using Abstract Syntax Tree (AST)-based Code2Vec representations, which, according to our results and findings from other studies [Liu et al., 2022a, Al Debeyan et al., 2022], prove to be very effective in capturing the hierarchical structure and relationships within the source code. This approach slightly improved the F1 score, yielding a precision of 0.72, a recall of 0.62, and an F1 score of 0.67, again using the Random Forest classifier.

While noteworthy, these results from Chapters 4 and 5 were based on a single project, Apache Tomcat 7, representing a within-project vulnerability prediction setting. This chapter is motivated by the need to evaluate our technique's performance in a mixed-project setting to determine its generalisability across multiple projects and gain insight into how much data-related challenges affect its performance, in other words, a stress test of the technique's generalisability. To achieve this, we used a dataset comprising code artefacts from several projects to assess the generalisability of the information retrieval-driven vulnerability prediction technique. This evaluation aims to understand how data-related factors affect the applicability of supervised learning techniques in vulnerability prediction across diverse software systems.

6.1.4 Research Question

To address the motivation outlined above, we pose the third and final research question of this thesis:

> *How well does the information retrieval-driven software vulnerability prediction technique generalise across multiple software systems?*

This research question is crucial for comprehensively evaluating our vulnerability prediction technique. While Chapters 4 and 5 focus on within-project assessments, this chapter examines mixed-project scenarios. This shift provides insights into the technique's generalisability from a stress test perspective, highlighting its performance across diverse software systems and the impact of data on vulnerability prediction.

6.1.5 Research Scope

The research scope of this chapter is as follows:

- **Programming Language:** We use a Java-based dataset.
- **Method-Level Vulnerability Prediction:** Our analysis focuses on method-level vulnerabilities. Vulnerabilities at the class or file level or outside the source code (e.g., configuration files, web services, or APIs) are not considered.
- **Mixed-Project Vulnerability Prediction:** We evaluate the information retrieval-driven vulnerability prediction technique using methods from multiple software projects.
- **Binary Classification:** The study focuses on binary classification, where a method is classified as vulnerable or non-vulnerable. Multi-class classification is not considered.

6.1.6 Significance and Contributions

This chapter contributes to software vulnerability prediction by evaluating the generalisability of our information retrieval-driven technique. It focuses on stress-testing our technique in a mixed-project vulnerability prediction setting and highlights the impact of data quality and quantity.

The study assesses the technique's generalisability across multiple projects, offering insights into its ability to predict vulnerabilities beyond the initial training project. The results will help software professionals understand the technique's capabilities, limitations, and applicability in within-project and mixed-project contexts. Additionally, it will provide insights into the impact of data quality and quantity on vulnerability prediction, highlighting the challenges and opportunities in this field.

6.1.7 Structure of the Chapter

Section 6.2 provides background information on mixed-project vulnerability prediction and data-related issues in vulnerability prediction. Section 6.3 outlines the study's methodology, including the dataset, data preprocessing, and other relevant aspects. Section 6.4 presents the experimental results. Section 6.5 discusses the results and their implications. Section 6.6 outlines the threats to validity for this chapter. Finally, Section 6.7 provides the conclusive answer to the third and final research question of this thesis.

6.2 Background

In the previous section, we discussed data-related issues in vulnerability prediction, noting that acquiring high-quality labelled data is a significant challenge for data-driven vulnerability prediction techniques. We mentioned how the insufficient training data challenge has led researchers to explore cross-project vulnerability prediction, which involves predicting vulnerabilities in a target software system using a model trained on a different source system.

This section provides some background on cross-project vulnerability prediction by briefly examining existing approaches. We highlight how researchers have approached this problem and the techniques they have used. Additionally, we address data quality challenges in vulnerability prediction, outlining the factors contributing to these challenges and their implications.

6.2.1 Cross-Project Vulnerability Prediction

Cross-project vulnerability prediction leverages data from a software project to predict vulnerabilities in another. This technique is appropriate when limited data is available for training a model. Data from various projects can be used to develop more robust models that predict vulnerabilities across a broader range of software systems. However, this approach is challenging due to differences in size, complexity, programming languages, and coding conventions between software projects. These differences make it challenging to develop a model that accurately predicts vulnerabilities across multiple projects. While within-project models have shown promising results, cross-project models often face performance issues [Kalouptsoglou et al., 2020, Siavvas et al., 2018].

Researchers have developed various machine and deep learning techniques for cross-project vulnerability prediction to address these challenges. Studies show that deep learning techniques generally outperform traditional machine learning methods [Kalouptsoglou et al., 2020] due to their ability to automatically construct high-level abstract feature representations of software systems, which is crucial for cross-project predictions [Liu et al., 2022b]. Thus, there is growing interest in deep learning for cross-project predictions.

Contemporary cross-project techniques often involve transferring ‘knowledge’ from one project to another to improve prediction accuracy. This involves two main concepts: transfer learning and domain adaptation. Transfer learning enhances a model by transferring knowledge from another domain [Weiss et al., 2016], where ‘domain’ refers to the data distribution, such as the distribution of vulnerable and non-vulnerable code samples. Knowledge is transferred from a source project to a target project to enhance prediction accuracy in the target project. Domain adaptation, a subset of transfer learning, involves adapting a model trained on a source domain to a target domain by reducing the distribution discrepancy between the two domains [Wilson and Cook, 2020]. It focuses on aligning the distributions of the source and target domains to improve prediction accuracy in the target domain.

Liu et al. [2022b] introduced the CD-VulD system, which utilises deep learning and domain adaptation to detect software vulnerabilities by learning token embeddings, constructing high-level representations, and mitigating domain divergence with the Metric Transfer Learning Framework (MTLF).

Nguyen et al. [2020] proposed the Dual Generator-Discriminator Deep Code Domain Adaptation Network (Dual-GD-DDAN). This GAN-based deep domain adaptation method enhances transfer learning between labelled and unlabelled projects, addressing mode collapse and improving predictive performance.

Kalouptsoglou et al. [2020] investigated the use of deep learning with software metrics to enhance cross-project vulnerability prediction, comparing machine learning models and assessing feature selection using a PHP dataset.

Nguyen et al. [2024] proposed a method that combines automatic representation learning and deep domain adaptation, utilising a cross-domain kernel classifier to enhance vulnerability detection in imbalanced labelled and unlabelled projects.

Zhang et al. [2023b] introduced CPVD, a cross-domain vulnerability detection method that utilises a code property graph and a Graph Attention Network with Convolution Pooling to extract features alongside Domain Adaptation Representation Learning to reduce distribution discrepancies.

Du et al. [2024] proposed CPMSVD, a method for snippet-level vulnerability detection using snippet attention, deep feature representation (AST-based Neural Network (ASTNN) for global, Bi-directional Gated Recurrent Unit (BiGRU) for local), domain adaptation (CORrelation ALignment (CORAL), Semi-Supervised Metric Transfer (SSMT)), and a K-Nearest Neighbors (KNN) classifier for vulnerability identification.

These methodologies demonstrate diverse and sophisticated approaches to cross-project vulnerability prediction, highlighting the complexity of the problem. The following subsection discusses how data quality and quantity are critical factors in vulnerability prediction.

6.2.2 Data Quality Challenges in Vulnerability Prediction

Data quality is crucial in vulnerability prediction because models adhere to the Garbage In, Garbage Out (GIGO) principle. While data-related issues affect both within- and cross-project vulnerability prediction settings, they are more pronounced in the latter due to the diversity of software systems. High-quality data is essential for accurate and reliable models [Jimenez et al., 2019], requiring significant attention to collection and processing [Zheng and Casari, 2018]. Vulnerability prediction requires samples of both vulnerable and non-vulnerable code, which compounds the data quality challenge [Walden et al., 2014]. Obtaining quality vulnerability data is difficult due to its infrequency [Zimmermann et al., 2010], inconsistent reporting [Anwar et al., 2021], and organisations' reluctance to share sensitive data [Coulter et al., 2020] related to the security posture of their software systems.

This subsection discusses critical factors contributing to data quality challenges in vulnerability prediction based on themes identified by Croft et al. [2022]: data generalisability, data accessibility, data preparation effort, data scarcity, label noise, and data noise.

6.2.2.1 Data Generalisability

Data generalisability is a significant challenge in vulnerability prediction. Real-world relevance is crucial, as synthetic data often used in vulnerability prediction analyses might not reflect actual vulnerabilities encountered in practice [Shahriar and Zulkernine, 2012]. External validity is also a concern, as data are often specific to particular programming languages, applications, or domains, limiting generalisability [Shahriar and Zulkernine, 2012, Hanif et al., 2021]. Additionally, vulnerabilities may span multiple components, and code representations used in predictions may fail to capture all relevant details, leading to a lack of completeness [Sidi et al., 2012, Shin and Williams, 2013, Tantithamthavorn et al., 2015].

6.2.2.2 Data Accessibility

Data accessibility is also a critical issue in vulnerability prediction. An aspect of this issue is the cold-start problem, a situation where a process must start without prior information. In our context, it refers to a situation characterised by a deficiency in previously identified vulnerabilities required for training a model [Croft et al., 2021]; thus, the limited data availability hinders comprehensive analyses [Neuhaus et al., 2007]. Also, data privacy concerns further complicate accessibility, as organisations may be reluctant to share sensitive data due to commercial, ethical, or legal reasons[1] [2] [3]. Even worse, when available, the data may be vague or incomplete due to inconsistent reporting practices [Anwar et al., 2021].

6.2.2.3 Data Preparation Effort

Collecting and labelling data is labour-intensive and requires significant human resources. It also demands domain expertise, as accurately labelling vulnerabilities requires deep knowledge of the concerned software system(s) [Zhang et al., 2023b].

[1] https://www.gov.uk/government/publications/data-sharing-governance-framework/data-sharing-governance-framework

[2] https://ico.org.uk/for-organisations/advice-for-small-organisations/whats-new/blogs/data-sharing-when-is-it-unlawful/

[3] https://www.ukri.org/wp-content/uploads/2021/08/MRC-0208212-GDPR-lawful-basis-research-consent-and-confidentiality.pdf

6.2.2.4 Data Scarcity

Data scarcity presents a significant challenge. There is often a significant imbalance, with vulnerable code samples being much rarer than non-vulnerable ones, which can skew analysis results. Additionally, the low sample size of vulnerable code limits the exposure of models to a diverse range of vulnerabilities and patterns, reducing the robustness of prediction models [Ban et al., 2019, Shu et al., 2022, Liu et al., 2019]. This particular issue is demonstrated in Chapters 4 and 5, where we had to rely on 'Candidate' vulnerable code samples in our vulnerability dataset to augment the limited number of confirmed vulnerable code samples (See Subsubsection 4.4.2.3) and also employ Synthetic Minority Oversampling Technique (SMOTE) to balance the number of vulnerable and non-vulnerable methods (See Subsubsection 4.4.3.6).

6.2.2.5 Label Noise

Label noise has a significant impact on the quality of vulnerability prediction. Incomplete labels are prevalent, with datasets containing dormant or latent vulnerabilities that remain undetected, resulting in data gaps [Bosu and MacDonell, 2013]. The absence or inaccuracy of vulnerability location information complicates analysis [He and Garcia, 2009], and misclassifying vulnerabilities can cause models to learn incorrect patterns, adversely affecting performance [Tantithamthavorn et al., 2015, Bosu and MacDonell, 2013]. This issue is one we highlighted in Subsubsection 4.7.1.2.

6.2.2.6 Data Noise

Data noise, particularly in source code, presents substantial challenges. Irrelevant noise, whether stylistic or syntactic, can obscure meaningful patterns [Leicht et al., 2017]. Redundant code elements, where identical or similar code exists across vulnerable and non-vulnerable artefacts, also hamper model performance [Tantithamthavorn et al., 2015]. Additionally, data heterogeneity from different sources, varying developer styles, and differing project conventions can reduce the versatility and effectiveness of prediction models, especially in cross-project settings [Tantithamthavorn et al., 2015, He and Garcia, 2009, Scandariato et al., 2014, Jimenez et al., 2016, Liu et al., 2022b, Stuckman et al., 2016]. Thus, normalisation techniques are often needed to mitigate these irregularities [Singh and Chaturvedi, 2020], which requires additional effort and expertise.

These data-related challenges adversely affect data-driven software engineering processes, such as vulnerability prediction. They emphasise the importance of employing robust data collection and processing techniques to ensure the development of reliable and accurate prediction models. Thus, addressing these challenges is essential for developing effective vulnerability prediction techniques. This chapter stress-tests our information retrieval-driven vulnerability prediction technique by evaluating its generalisability across multiple software systems, focusing on data quality and quantity limitations. The subsequent sections outline the methodology, results, and discussion of this study.

6.3 Methodology

This section outlines the methodology used in this chapter. We first provide an overview of the methodology, followed by detailed descriptions of the dataset used in the experiments, data preprocessing techniques, information retrieval strategies, and machine learning analyses. These machine learning analyses use metrics data developed using token-based and Code2Vec-based representations.

6.3.1 Overview of the Methodology

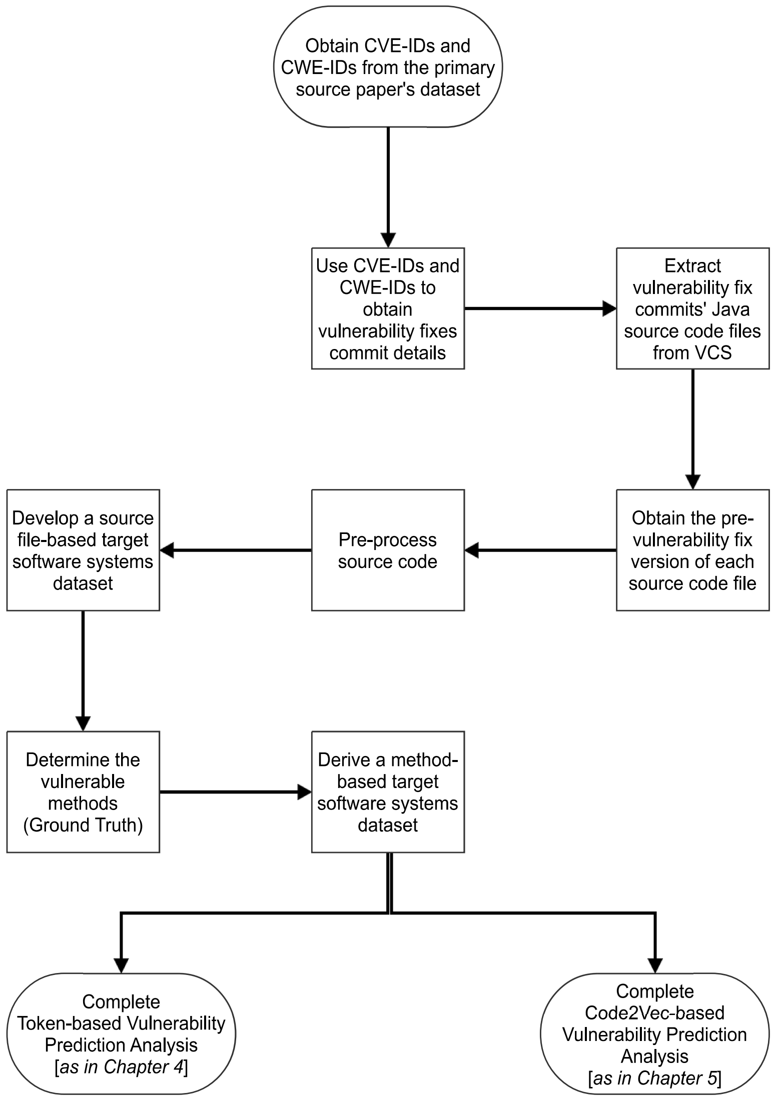

Obtain CVE-IDs and CWE-IDs from the primary source paper's dataset
Use CVE-IDs and CWE-IDs to obtain vulnerability fixes commit details
Extract vulnerability fix commits' Java source code files from VCS
Obtain the pre-vulnerability fix version of each source code file
Pre-process source code
Develop a source file-based target software systems dataset
Determine the vulnerable methods (Ground Truth)
Derive a method-based target software systems dataset
Complete Token-based Vulnerability Prediction Analysis [as in Chapter 4]
Complete Code2Vec-based Vulnerability Prediction Analysis [as in Chapter 5]

Figure 6.1: Token- and Code2Vec-Based Vulnerability Prediction Methodology Overview (Mixed-Project)

Figure 6.1 illustrates the overall methodology used in this study, divided into three core components: dataset acquisition for target software systems, token-based vulnerability prediction analysis, and Code2Vec-based vulnerability prediction analysis.

The first eight steps of the flowchart relate to the dataset acquisition process for the target software systems. After completing dataset acquisition (step eight), the flowchart branches into two separate paths for token-based and Code2Vec-based vulnerability prediction analyses, as these analyses are conducted concurrently and independently.

This chapter's token-based vulnerability prediction analysis comprises five core stages: extraction of token representations, generation of n-grams (shingles) from tokens, setup of information retrieval, development of metrics data, and machine learning analysis.

This chapter's Code2Vec-based vulnerability prediction analysis comprises four core stages: extraction of Code2Vec representations, setup of information retrieval, development of metrics data, and machine learning analysis.

For detailed token-based and Code2Vec-based vulnerability prediction methodologies, refer to Sections 4.4 and 5.4, respectively.

6.3.1.1 Part A: Target Software Systems Dataset Acquisition

The process of acquiring the target software systems dataset involved six phases:

1. **Collection of CVE and CWE IDs:** The primary source paper for the dataset used in this chapter provided several thousand samples of vulnerable and non-vulnerable code from various software systems, identified by their associated CVE and CWE IDs We compiled a list of these IDs to identify the vulnerable code samples[4].

2. **Identification of Vulnerability-Fix Commits:** The secondary reference paper, from which the primary source paper obtained the dataset, contained detailed information about the code samples, including vulnerability-fix com-

[4]`https://cve.mitre.org/`

mits linked to the CVE and CWE IDs. We cross-referenced these IDs with those in the dataset to identify the relevant commits in the respective version control repositories.

3. **Retrieval of Source Code:** We created a 'file-level' dataset by retrieving Java source code files associated with the identified vulnerability-fix commits. By 'file-level', we mean in the dataset context, each file represents a single code artefact, i.e., the unit of analysis. We also retrieved previous versions of each source code file prior to the fixes, including any files that were deleted as part of the fixes.

4. **Preprocessing of Source Code:** We removed comments and other irrelevant elements from the source code files to eliminate noise that could affect the analysis.

5. **Extraction of Ground Truth Data:** Using the current and previous versions of the source code files, we determined the ground truth for each method, labelling them as either vulnerable or non-vulnerable.

6. **Construction of the Target Software Systems Dataset:** We parsed the source code files and constructed a method-level dataset comprising methods labelled as vulnerable or non-vulnerable based on the ground truth data.

6.3.1.2 Part B: Token-Based Analysis

This part of the methodology is identical to the token-based vulnerability prediction analysis detailed in Section 4.4, specifically from Subsubsections 4.4.1.1 to 4.4.1.6. However, we note that churn-related metrics were excluded in this chapter due to the dataset's multi-system nature.

6.3.1.3 Part C: Code2Vec-Based Analysis

This part of the methodology mirrors the Code2Vec-based vulnerability prediction analysis detailed in Section 5.4, specifically from Subsubsections 5.4.1.1 to 5.4.1.5. Again, churn-related metrics were excluded in this chapter due to the dataset's multi-system nature.

This section provided an overview of the methodology, summarising the experimental procedures used in this chapter. The upcoming sections provide more details on the dataset and other relevant aspects of the methodology.

6.3.2 Dataset

We obtained our dataset in the first quarter of 2024 from Al Debeyan et al. [2022]'s work (reviewed in Subsection 3.2.1). The dataset is a CSV file containing information on several Java methods from multiple software systems. It includes columns holding information on the project name, CVE ID, CWE ID, method name, N-grams of AST representations, and a binary vulnerability status column indicating whether the method is vulnerable or not.

Since our analyses required the source code of the methods, which was not included in the dataset, we retrieved it from the software systems' version control repositories. We referred to the study by Reis and Abreu [2021], from which Al Debeyan et al. [2022] obtained their dataset. This secondary source study provided detailed information about the code samples, including the vulnerability-fix commits linked to the CVE and CWE IDs. This enabled us to identify the relevant commits in the version control repository (GitHub) and retrieve the source code of the methods.

6.3.2.1 Target Software Systems

To retrieve the source code of the methods, we used PyDriller[5], a Python framework for extracting information from version control repositories. We cross-referenced the CVE and CWE IDs in Al Debeyan et al. [2022]'s dataset with those in Reis and Abreu [2021]'s dataset to identify the vulnerability-fix commits.

This allowed us to retrieve the source code of the methods associated with the vulnerability-fix commits. We also obtained previous versions of each source code file before the fixes. Additionally, we gathered supplementary information for each file, including the project name, commit hash, GitHub URL, and file paths. This enabled us to build a file-centric dataset of target software systems, primarily

[5]`https://pydriller.readthedocs.io/`

comprising pre- and post-vulnerability-fix versions of the source code files associated with the vulnerability-fix commits for the CVE and CWE IDs obtained in our primary source paper's dataset.

6.3.2.2 Ground Truth Data Development

Since our vulnerability prediction analyses in this thesis focus on the method level, we shifted from file-level to method-level analysis to determine the ground truth data for each method.

We compared the pre- and post-vulnerability-fix versions of the source code files to label each method as vulnerable or non-vulnerable based on the changes made. A method was considered vulnerable if it existed in the pre-fix version but was deleted or modified in the post-fix version. Conversely, a method was considered non-vulnerable if it remained unchanged between the two versions or was added in the post-fix version.

Apart from dataset acquisition, this ground truth determination approach is the only methodological similarity between our work and the primary source paper, i.e., Al Debeyan et al. [2022]'s work. The primary source paper adopted the same approach to determine the ground truth in their methodology.

6.3.2.3 National Institute of Standards and Technology (NIST) Software Assurance Reference Dataset (SARD)

Building on previous chapters, this work also uses the NIST Software Assurance Reference Dataset as a source of known Java software vulnerabilities. See Subsubsection 4.4.2.3.

We used the same SARD artefacts as in previous chapters, totalling 20,692 known vulnerable methods (see Table 4.9). To ensure a fair comparison between within-project and mixed-project techniques, we preprocessed the source code files in the vulnerability dataset in the same manner as the target software systems dataset, as described in Subsubsection 6.3.1.1. This included removing irrelevant elements, such as comments, to reduce noise in the analysis.

Table 6.1: Dataset Details - Multiple Software Systems

Description	**Value**
Total Number of Software Systems	132
Total Number of Vulnerability Fixes	672
Total Number of Indexed Vulnerability Dataset Methods	20,692

Table 6.1 provides an overview of the dataset details for the multiple software systems used in this chapter. The target software systems dataset included 132 software systems with 672 vulnerability fixes. The vulnerability dataset contained 20,692 known vulnerable methods, consistent with the number of known vulnerable methods used in Chapters 4 and 5.

6.3.3 Data Preprocessing

We used the same data preprocessing techniques as in the previous two chapters. For the token-based analysis, we followed the method in Chapter 4 (Subsection 4.4.3). For the Code2Vec-based analysis, we applied the technique described in Chapter 5 (Subsection 5.4.3).

6.3.3.1 Data Deduplication

Similar to the previous two chapters, we removed duplicate methods (based on their representations) to prevent data leakage in the machine learning analysis. Our deduplication process is fully described in Subsubsection 4.4.3.3.

Failing to address data leakage in vulnerability prediction analyses adequately can lead to overly optimistic results. This is an issue we observed in Al Debeyan et al. [2022]'s study, the primary source paper for this chapter's dataset. We reviewed the study in Subsection 3.2.1 and further discussed it in Subsection 5.6.1. In their methodology, they stated that they removed duplicates from the dataset. However, a closer look at their dataset revealed that their deduplication process was flawed. Their binary classification dataset is a CSV file comprising six columns, including project name, CVE ID, CWE ID, method name, AST N-grams, and vulnerability status, which contain 53,201 instances. Upon inspection, we observed that only 29,693 AST N-grams (features) are unique out of these instances, which accounts for 55.81% of the dataset. Furthermore, the study employed a 10-fold cross-validation, which means that the dataset was split into ten folds, and the model was trained and tested ten times. This means that the same instances were used in multiple folds, likely leading to data leakage. We observed that they conducted their deduplication using all the CSV columns rather than just the AST N-grams column. This would have led to some of the exact AST N-grams appearing in multiple folds, potentially resulting in overly optimistic results. This is a significant issue in the study, affecting the reliability of the results.

Table 6.2: Post-Deduplication Dataset Details - Multiple Software Systems

Description	**Value**	**%**
Number of Non-Vulnerable Methods Post-Deduplication	19,421	94.72
Number of Vulnerable Methods Post-Deduplication	1,081	5.28
Total Number of Methods Post-Deduplication	20,502	

Table 6.2 provides an overview of the post-deduplication dataset details for the token-based and Code2Vec-based analyses. The dataset comprises 20,502 methods, including 19,421 non-vulnerable (94.72%) and 1,081 vulnerable (5.28%). The class distribution is significantly imbalanced, with non-vulnerable methods vastly outnumbering vulnerable ones.

6.3.3.2 Feature Scaling

As in the previous chapters, we normalised the metrics data using Min-Max scaling to ensure uniform scaling across all feature values and prevent bias towards features with larger values. Subsubsection 4.4.3.5 provides a detailed explanation of this process.

6.3.3.3 Software System Vulnerabilities and Data Imbalance

After deduplication, we observed a significant data imbalance in our token-based and Code2Vec-based datasets, with non-vulnerable methods vastly outnumbering vulnerable ones, which was expected. This issue is common in vulnerability prediction research, as discussed in Subsubsection 6.2.2.4.

Typically, software systems contain far more non-vulnerable than vulnerable code artefacts, making it challenging to train an effective classifier. This imbalance can bias classifiers towards the majority class (non-vulnerable), reducing performance in predicting the minority class (vulnerable).

Although the imbalance in our datasets was not as extreme as in the previous two chapters, it was still significant, with non-vulnerable methods at 94.72% and vulnerable methods at 5.28%, as shown in Table 6.2. To address this, we employed the SMOTE as in Chapters 4 and 5. See Subsubsection 4.4.3.6 for details.

6.3.4 Information Retrieval

We constructed an information retrieval document index using 20,692 source code token representations of methods in the vulnerability dataset, as shown in Table 6.1. We created two information retrieval setups using Apache Lucene, one for token-based and one for Code2Vec-based analysis.

Both setups are fully described in the previous chapters. Refer to Subsections 4.4.4 and 5.4.4 for the token-based analysis and Code2Vec-based analysis, respectively.

Table 6.3: Percentage of Vulnerable versus Non-Vulnerable Methods with Hits - Multiple Software Systems

Number of Vulnerable Methods	1081
Number of Vulnerable Methods with Hits	914
% of Vulnerable Methods with Hits	84.55
Number of Non-Vulnerable Methods	19,421
Number of Non-Vulnerable Methods with Hits	8,989
% of Non-Vulnerable Methods with Hits	46.28

Table 6.3 supplements Table 6.2. It shows the percentage of vulnerable and non-vulnerable methods with hits in the target software systems dataset, demonstrating that vulnerable methods are significantly more likely to share patterns with vulnerable code than non-vulnerable methods, an observation consistent with the previous chapters and extensively discussed in Subsections 4.6.2 and 5.6.2.

6.3.5 Machine Learning Analysis

We proceeded to the machine learning analysis phase, following the token-based and Code2Vec-based information retrieval and metrics data development phases. The metrics data abstracts the type of representation used in the information retrieval phase. The metrics data are the independent variables for the machine learning classification task, while the ground truth data is the dependent variable. The machine learning analysis described in Subsection 4.4.5 applies to Chapters 4, 5, and this chapter.

The only difference in this chapter is the exclusion of churn-related metrics from the metrics data development phase, as the dataset is derived from multiple software systems, not multiple releases of the same system. We excluded TICC, TRICC, NTDT, and NTDDT for the token-based analysis (see Subsection 4.3.1 and Table 4.1). For the Code2Vec-based analysis, we excluded PICC, PRICC, NTDP, and NTDDP (see Subsection 5.3.1 and Table 5.1).

Apart from these differences, the machine learning analysis phase in this chapter is similar to that of the previous two chapters.

6.3.6 Approach to Question 3

The primary goal of this chapter is to address Research Question 3: *How well does the information retrieval-driven software vulnerability prediction technique generalise across multiple software systems?*

We set four objectives to address this question:

1. Identify the most suitable classifier for vulnerability prediction across multiple software systems for token-based and Code2Vec-based analyses.

2. Identify the best-performing combination of software metrics for vulnerability prediction across multiple software systems for token-based and Code2Vec-based analyses.

3. Evaluate the impact of hyperparameter tuning on the performance of the best-performing classifier for token-based and Code2Vec-based analyses.

4. Investigate the impact of dataset variability on vulnerability prediction performance.

6.3.6.1 Objective 1

To achieve the first objective, we assessed the performance of ten machine learning classifiers on the token-based and Code2Vec-based metrics data using precision, recall, and F1 scores. The classifier with the highest F1 score, the harmonic mean of precision and recall, was selected.

6.3.6.2 Objective 2

We used Sequential Forward Selection for the second objective to find the optimal combination of metrics for vulnerability prediction. This technique systematically evaluates different metrics combinations to identify the best feature set for classification (see Subsubsection 4.4.6.2 for details).

6.3.6.3 Objective 3

To achieve the third objective, we employed the Grid Search technique to tune the hyperparameters of the best-performing classifier. This technique systematically evaluates different hyperparameter combinations to find the optimal configuration (see Subsubsection 4.4.6.3 for details).

6.3.6.4 Objective 4

The dataset used in this chapter is derived from multiple software systems, introducing variability that can affect vulnerability prediction. For the fourth objective, we investigated the impact of dataset variability on vulnerability prediction performance. We conducted analyses to investigate the impact of dataset variability on vulnerability prediction by examining the Coefficient of Variation of the metrics used in the analyses.

The Coefficient of Variation measures a dataset's relative variability. It is a dimensionless quantity expressed as a percentage, making it helpful in comparing the variability of datasets with different units or means. The Coefficient of Variation is calculated by dividing a dataset's standard deviation by its mean and multiplying by

100. ‘Dataset’ refers to a single metric used in the vulnerability prediction analysis in this context.

The primary advantages of using the Coefficient of Variation include:

- **Independence from Units:** As a dimensionless measure, the Coefficient of Variation allows direct comparisons between datasets with different units.
- **Relative Measure:** Coefficient of Variation provides a relative measure of variability, which is helpful in comparing datasets with different means.
- **Ease of Interpretation:** Expressed as a percentage, Coefficient of Variation is easy to interpret.
- **Simplicity of Calculation:** Coefficient of Variation requires only the mean and standard deviation of the dataset.
- **Facilitates Comparisons:** Coefficient of Variation simplifies the comparison of variability between multiple datasets.

By calculating the Coefficient of Variation of the metrics used in our vulnerability prediction analyses, we gained insights into the stability and consistency of the metrics across the datasets, allowing us to understand the impact of dataset variability on vulnerability prediction. We conducted these analyses for our single software system (within-project dataset) used in Chapters 4 and 5, as well as for the multiple software systems (mixed-project dataset) used in this chapter, for both token-based and Code2Vec-based metrics.

6.3.7 Summary of Methodological Differences Across Chapters

To conclude this methodology section, we summarise the key methodological differences between this chapter and the previous two chapters in Table 6.4. The table highlights differences in the prediction setting, dataset type (software system), number of metrics, source code attributes considered, and the type of representation used in the information retrieval phase.

In the subsequent sections, we present the results of the vulnerability prediction analysis, discuss the findings, and address the threats to validity associated with

Table 6.4: Methodological Differences Between Chapters 4, 5, and the Current Chapter

Description	**Chapter 4**	**Chapter 5**	**Current Chapter**
Setting	Within-Project	Within-Project	Mixed-Project
Software System	Single	Single	Multiple
Number of Metrics	Sixteen (16)	Sixteen (16)	Twelve (12)
Code Pattern Similarity	Applicable	Applicable	Applicable
Intricacy	Applicable	Applicable	Applicable
Size	Applicable	Applicable	Applicable
Code Churn	Applicable	Applicable	Not Applicable
Representation	Token-Based	Code2Vec-Based	Both

this chapter. We then directly and conclusively address the third and final research question in this thesis.

6.4 Results

This section presents the results of the vulnerability prediction analysis conducted in this chapter. We present the results according to the four objectives outlined in Subsection 6.3.6.

6.4.1 Objective 1 Results

Identify the most suitable classifier for vulnerability prediction across multiple software systems for token-based and Code2Vec-based analyses.

This objective aimed to identify the most effective classifier for predicting vulnerabilities across multiple software systems, using both token-based and Code2Vec-based analyses.

6.4.1.1 Token-Based Evaluation Metrics Trend Analysis

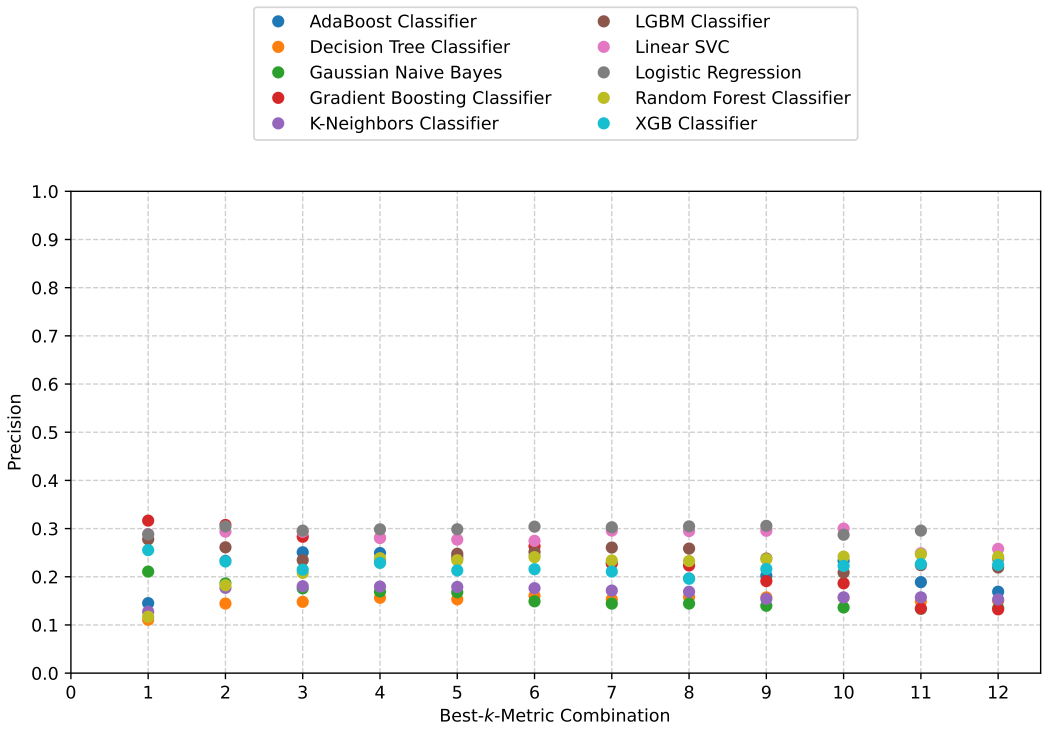


Figure 6.2: Precision Trend across all Best-*k*-Performing Metrics Combinations (Token-Based Analysis)

Figure 6.2 shows the precision trend for the top *k* metrics combinations in the token-based analysis. Precision values for all classifiers are notably low, with the highest being approximately 0.32, achieved by the Gradient Boosting classifier at

$k = 1$. All other precision values for the classifiers and top k combinations are below this.

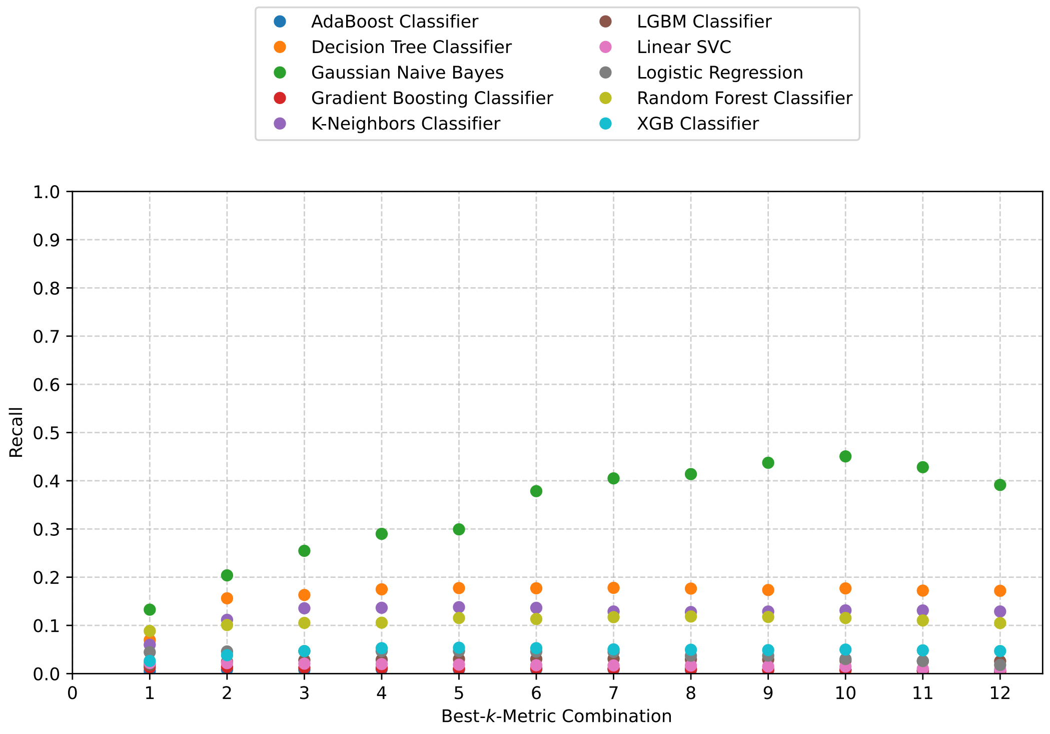


Figure 6.3: Recall Trend across all Best-k-Performing Metrics Combinations (Token-Based Analysis)

Figure 6.3 shows the recall trend for the top k metrics combinations in the token-based analysis. Similar to the precision trend, recall values are relatively low for all classifiers across all top k combinations. The only exception is the Gaussian Naïve Bayes classifier at $k = 10$, achieving a recall of 0.45. Although this is the highest recall value, it remains relatively low. All other recall values from other classifiers are below 0.20.

Figure 6.4 shows the F1 score trend for the token-based analysis's top k metrics combinations. Since the F1 score is the harmonic mean of precision and recall, low F1 scores are expected for all classifiers. The only classifier exceeding the 0.20 threshold was the Gaussian Naïve Bayes, maintaining this level from $k = 3$ to $k = 12$, with a maximum F1 score of 0.22. Other classifiers had significantly lower F1 scores across all top k combinations.

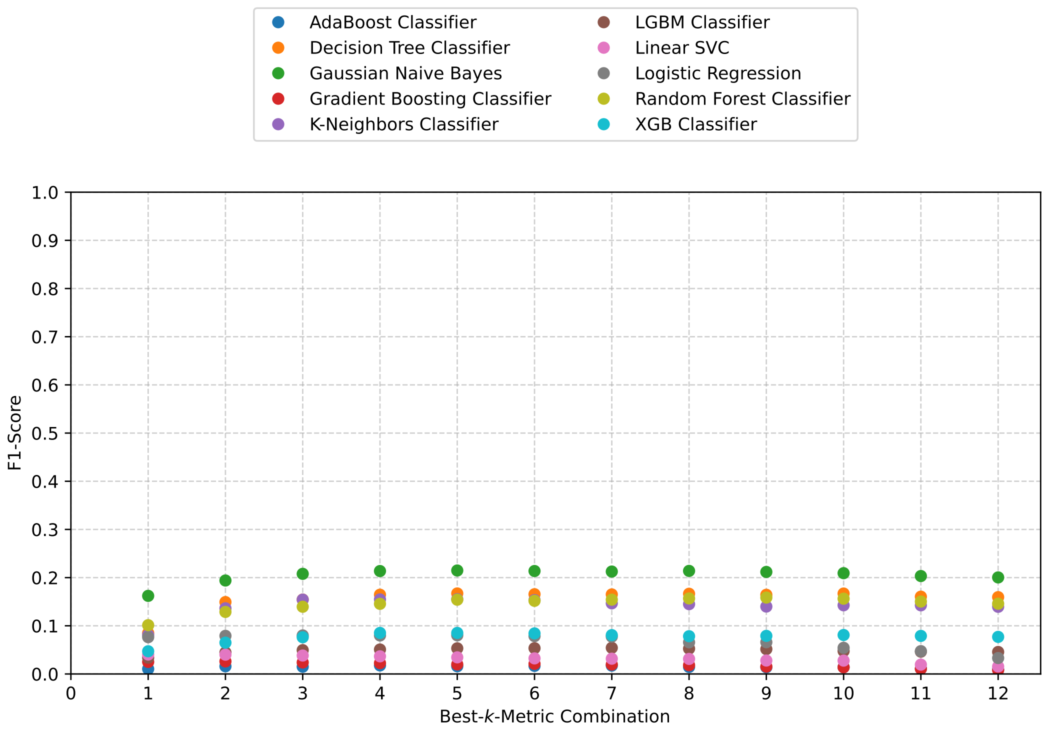


Figure 6.4: F1 score Trend across all Best-*k*-Performing Metrics Combinations (Token-Based Analysis)

6.4.1.2 Code2Vec-Based Evaluation Metrics Trend Analysis

Figure 6.5 illustrates the precision trend for the top k metrics combinations in the Code2Vec-based analysis. Similar to the token-based analysis, precision values are low for all classifiers. The Linear Support Vector classifier achieved the highest precision at $k = 1$, $k = 2$, and $k = 12$, with values around 0.31. The Logistic Regression classifier outperformed it from $k = 3$ to $k = 11$, maintaining a precision of around 0.30. Other classifiers had significantly lower precision values.

Figure 6.6 illustrates the recall trend for the top k metrics combinations in the Code2Vec-based analysis. Recall values are low for all classifiers. The Gaussian Naïve Bayes classifier performed best, achieving the highest recall of around 0.26 at $k = 12$. Other classifiers had recall values below 0.20, performing significantly worse.

Figure 6.7 shows the F1 score trend for the top k metrics combinations in the Code2Vec-based analysis. F1 scores are low for all classifiers. The Gaussian Naïve

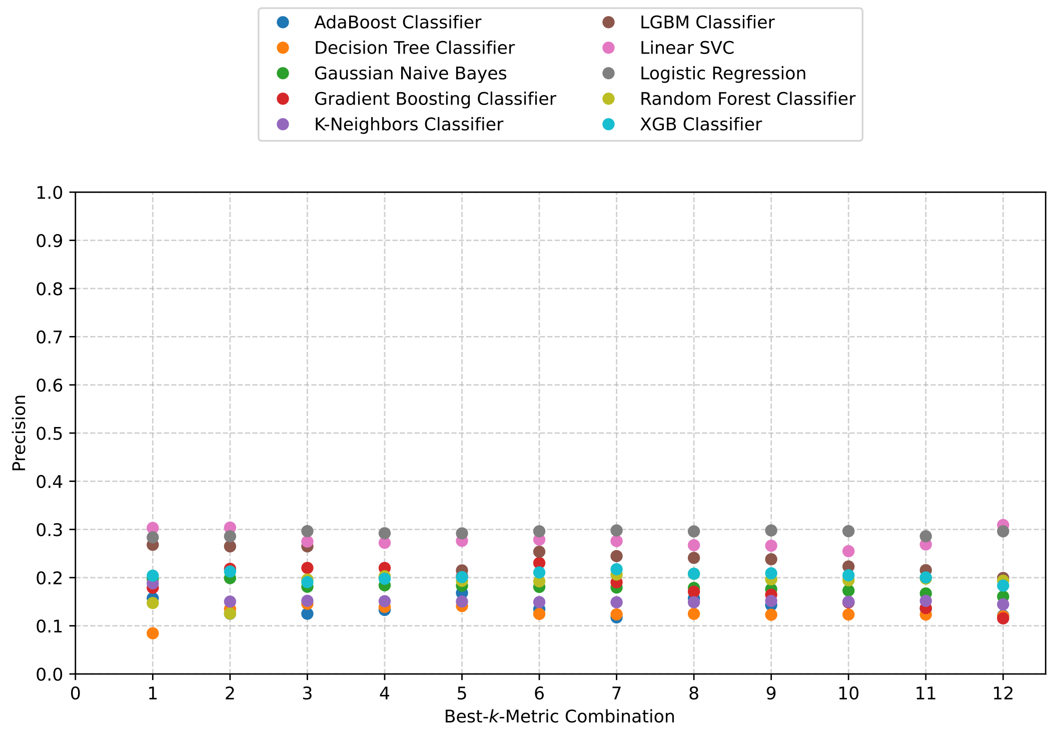


Figure 6.5: Precision Trend across all Best-k-Performing Metrics Combinations (Code2Vec-Based Analysis)

Bayes classifier achieved the highest F1 score, around 0.20 at $k = 8$. Other classifiers had F1 scores below 0.20, performing significantly worse.

To address the first objective, we identified the Gaussian Naïve Bayes classifier as the best-performing classifier for vulnerability prediction across multiple software systems in both the token-based and Code2Vec-based analyses. However, we highlight that the performance of all classifiers across the top k metrics combinations was generally poor and not actionable, with low precision, recall, and F1 scores.

6.4.2 Objective 2 Results

Identify the best-performing combination of software metrics for vulnerability prediction across multiple software systems for token-based and Code2Vec-based analyses.

This objective aimed to identify the optimal combination of metrics for vulnerability prediction across multiple software systems using Sequential Forward Selec-

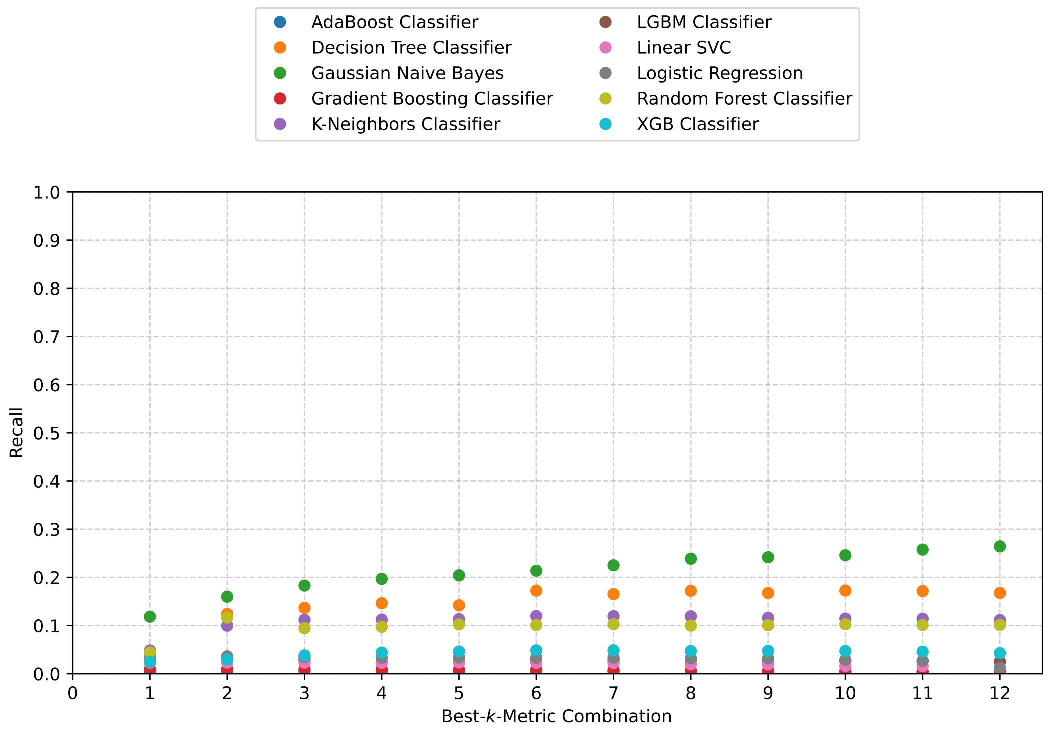


Figure 6.6: Recall Trend across all Best-*k*-Performing Metrics Combinations (Code2Vec-Based Analysis)

tion for both token-based and Code2Vec-based analyses. Similar to Chapters 4 and 5, we first present the Correlation Matrix for both the token-based and Code2Vec-based analyses in Figures 6.8 and 6.9, respectively. The matrices feature a two-level grouping structure. Firstly, rows and columns are categorised into ground truth, hit-independent, and hit-dependent metrics. Secondly, the latter two categories are further clustered based on their constituent metrics. The Pearson Correlation Coefficient quantifies the linear association between two variables, ranging from -1 (strong negative correlation) to 1 (strong positive correlation), with 0 indicating no correlation. We categorise the values within the correlation matrix according to the classification scheme presented in Table 4.13. Each matrix includes a colour scale on the right side, using a diverging colour scheme: red hues at the top indicate a strong positive correlation, white in the middle signifies no correlation, and teal tones at the bottom indicate a strong negative correlation.

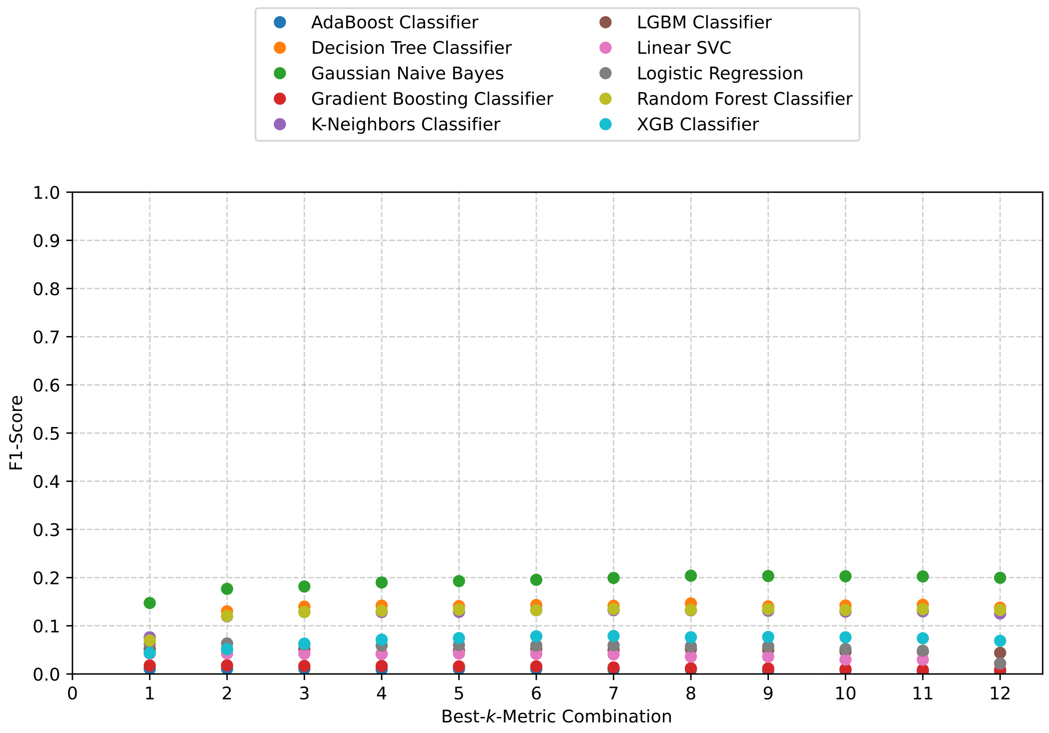


Figure 6.7: F1 score Trend across all Best-*k*-Performing Metrics Combinations (Code2Vec-Based Analysis)

Below, we outline the critical observations derived from the correlation matrix. The complete names of the metrics are provided in Tables 5.1 and 5.3, with metrics abbreviated in the correlation matrix for brevity.

6.4.2.1 Token-Based Metrics Correlation Analysis

Figure 6.8 presents the correlation matrix of all token-based metrics and the ground truth.

The main observations are as follows:

1. Certain cells within Band No. 1 exhibit values of ‘1’ or values close to ‘1’, indicating a strong positive correlation between specific metrics. Notably, there is a significant correlation between NUSM and SMR, as well as between metrics and their distinct equivalents, such as NHS and NDHS. This outcome aligns with expectations due to the conceptual similarities among these metrics. However, these high correlations suggest potential redundancy, which may affect model performance.

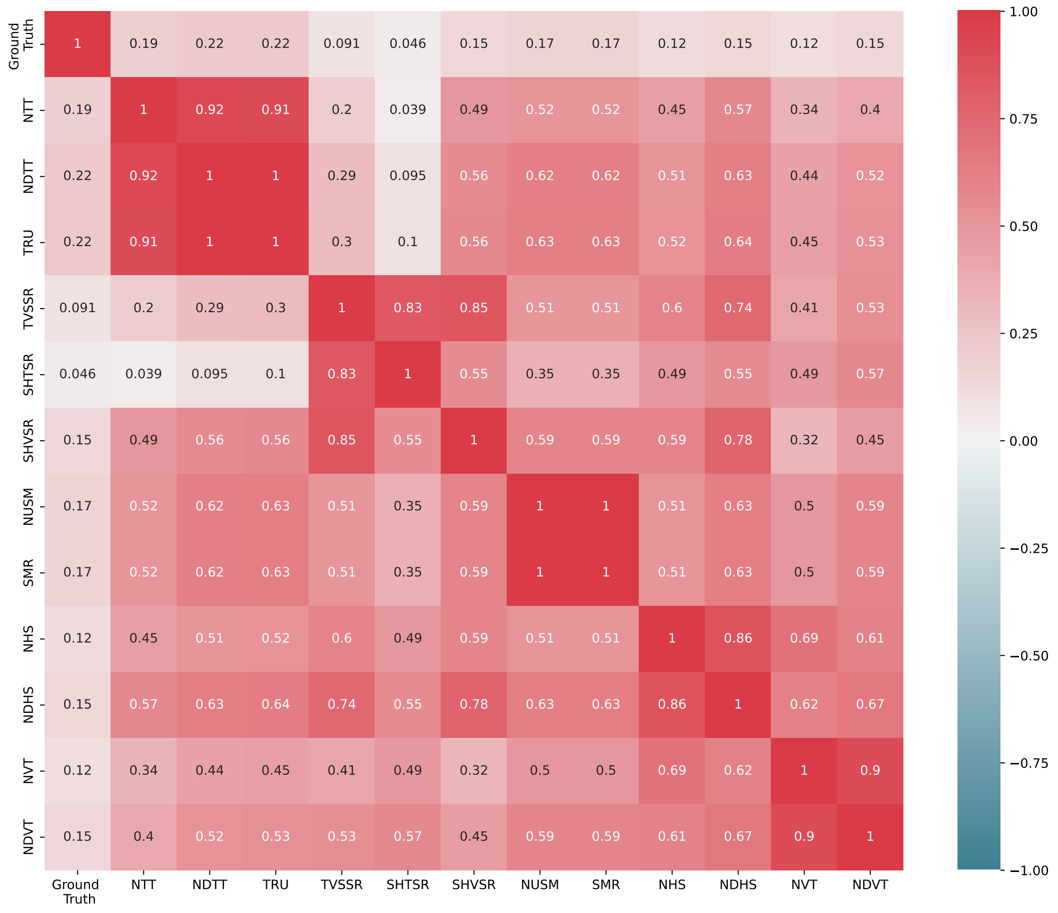


Figure 6.8: Correlation Matrix of all Metrics + Ground Truth (Token-Based Analysis)

2. No metrics exhibit negative correlations with other metrics or the ground truth, as no cells have a value below '0.' The lowest value is '0.039', observed between SHTSR and NTT, which still indicates a weak positive correlation.

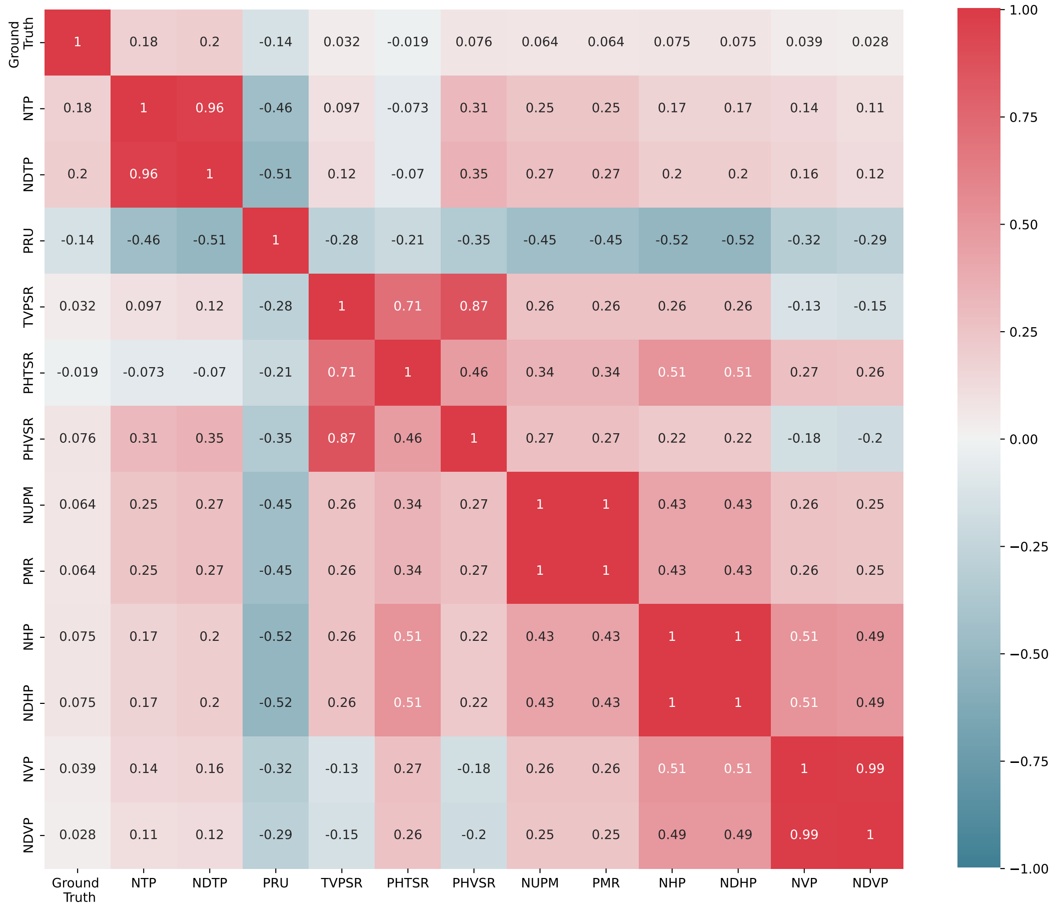
3. The NTT & NVT metrics, tailored to the target software system and the vulnerability dataset, respectively, show a weak positive correlation of 0.34 (Band No. 4). This correlation improves to 0.55 (Band No. 3) when hits are considered, as seen between SHTSR & SHVSR. This highlights the importance of hits in the vulnerability prediction process.

4. The TVSSR metric, encompassing the target software system and its corresponding method in the vulnerability dataset, shows a stronger association with SHTSR & SHVSR than between SHTSR & SHVSR alone. TVSSR has a high positive correlation of 0.83 (Band No. 1) with SHTSR and 0.85 (Band

No. 1) with SHVSR, compared to 0.55 (Band No. 3) between SHTSR & SHVSR. This highlights the significance of the TVSSR metric in vulnerability prediction.

5. The ground truth shows no strong correlation with any single metric, with the highest being 0.22 (Band No. 4) between the ground truth and NDTT & TRU.

6.4.2.2 Code2Vec-Based Metrics Correlation Analysis



	Ground Truth	NTP	NDTP	PRU	TVPSR	PHTSR	PHVSR	NUPM	PMR	NHP	NDHP	NVP	NDVP
Ground Truth	1	0.18	0.2	-0.14	0.032	-0.019	0.076	0.064	0.064	0.075	0.075	0.039	0.028
NTP	0.18	1	0.96	-0.46	0.097	-0.073	0.31	0.25	0.25	0.17	0.17	0.14	0.11
NDTP	0.2	0.96	1	-0.51	0.12	-0.07	0.35	0.27	0.27	0.2	0.2	0.16	0.12
PRU	-0.14	-0.46	-0.51	1	-0.28	-0.21	-0.35	-0.45	-0.45	-0.52	-0.52	-0.32	-0.29
TVPSR	0.032	0.097	0.12	-0.28	1	0.71	0.87	0.26	0.26	0.26	0.26	-0.13	-0.15
PHTSR	-0.019	-0.073	-0.07	-0.21	0.71	1	0.46	0.34	0.34	0.51	0.51	0.27	0.26
PHVSR	0.076	0.31	0.35	-0.35	0.87	0.46	1	0.27	0.27	0.22	0.22	-0.18	-0.2
NUPM	0.064	0.25	0.27	-0.45	0.26	0.34	0.27	1	1	0.43	0.43	0.26	0.25
PMR	0.064	0.25	0.27	-0.45	0.26	0.34	0.27	1	1	0.43	0.43	0.26	0.25
NHP	0.075	0.17	0.2	-0.52	0.26	0.51	0.22	0.43	0.43	1	1	0.51	0.49
NDHP	0.075	0.17	0.2	-0.52	0.26	0.51	0.22	0.43	0.43	1	1	0.51	0.49
NVP	0.039	0.14	0.16	-0.32	-0.13	0.27	-0.18	0.26	0.26	0.51	0.51	1	0.99
NDVP	0.028	0.11	0.12	-0.29	-0.15	0.26	-0.2	0.25	0.25	0.49	0.49	0.99	1

Figure 6.9: Correlation Matrix of all Metrics + Ground Truth (Code2Vec-Based Analysis)

Figure 6.9 presents the correlation matrix of all Code2Vec-based metrics and the ground truth.

The main observations are:

1. Compared to the token-based correlation matrix (Figure 6.8), the Code2Vec-based correlation matrix generally shows weaker positive correlations between metrics, as evidenced by the overall lighter hues.

2. Similar to the token-based correlation matrix, the Code2Vec-based matrix reveals a significant positive correlation between specific metrics. For example, NUPM & PMR and NHP & NDHP exhibit a perfect positive correlation of 1 (Band No. 1). These correlations suggest that these metrics assess the same attributes with slight methodological differences, indicating potential redundancy.

3. Unlike the token-based correlation matrix, the Code2Vec-based matrix exhibits negative correlations between specific metrics, particularly between PRU and other metrics, including the ground truth. The highest negative correlations are between PRU and NHP and between PRU and NDHP, with a value of -0.52 (Band No. 9). This is expected as PRU measures the uniqueness of a method, which inversely relates to the number of hits (NHP & NDHP).

4. The NTP & NVP metrics, analogous to the NTT & NVT metrics in the token-based analysis, focus on the target software system and the vulnerability dataset, respectively. These metrics show a very weak positive correlation of 0.14 (Band No. 5). When hits are incorporated, such as between PHTSR and PHVSR, the correlation improves to 0.46 (Band No. 3).

5. The TVPSR metric, which integrates the target software system and its corresponding most similar method in the vulnerability dataset, shows a stronger association with PHTSR and PHVSR than the direct association between PHTSR and PHVSR. TVPSR exhibits high positive correlations of 0.71 (Band No. 2) with PHTSR and 0.87 (Band No. 1) with PHVSR, whereas the correlation between PHTSR and PHVSR is 0.46 (Band No. 3). This highlights the significance of the TVPSR metric in vulnerability prediction.

6. Similar to the token-based correlation matrix and previous chapters, the ground truth has no strong correlation with any single metric, with the highest being 0.20 (Band No. 4) between the ground truth and NDTP.

In Chapters 4 and 5, we discussed how the correlation matrix can help identify and exclude redundant metrics to improve a model's performance. However, we observed that excluding seemingly redundant metrics often did not enhance performance and sometimes slightly worsened it. Therefore, we included all metrics in our analyses to ensure no potentially beneficial metrics were excluded. Instead, we used the Sequential Forward Selection technique to identify the best-performing combination of metrics for vulnerability prediction. We adopted the same approach in this chapter for consistency.

6.4.2.3 Token-Based Classifier Performance Analysis

Table 6.5: Best Performance Per Classifier (Token-Based Analysis)

Classifier	**Best k**	**Precision**	**Recall**	**F1 score**
Gaussian Naïve Bayes	5	0.16780	0.29921	0.21477
Decision Tree classifier	5	0.15315	0.17742	0.16714
Random Forest classifier	9	0.23641	0.11748	0.15860
K-Nearest Neighbors classifier	5	0.17887	0.13784	0.15525
XGBoost classifier	5	0.21315	0.05374	0.08520
Logistic Regression	5	0.29859	0.04681	0.08022
LightGBM classifier	7	0.26069	0.03062	0.05428
Linear Support Vector classifier	1	0.28867	0.02193	0.04035
Gradient Boosting classifier	2	0.30755	0.01378	0.02611
AdaBoost classifier	7	0.22889	0.00907	0.01740

Table 6.5 presents the best performance per classifier in the token-based analysis, sorted by the highest F1 score in descending order. The Gaussian Naïve Bayes classifier achieved the highest F1 score of 0.21477 at $k = 5$, followed by the Decision Tree classifier with an F1 score of 0.16714 at $k = 5$, and the Random Forest classifier with an F1 score of 0.15860 at $k = 9$.

At the lower end, the AdaBoost classifier achieved the lowest F1 score of 0.01740 at $k = 7$, followed by the Gradient Boosting classifier with an F1 score of 0.02611 at $k = 2$, and the Linear Support Vector classifier with an F1 score of 0.04035 at $k = 1$.

6.4.2.4 Code2Vec-Based Classifier Performance Analysis

Table 6.6: Best Performance Per Classifier (Code2Vec-Based Analysis)

Classifier	Best k	Precision	Recall	F1 score
Gaussian Naïve Bayes	8	0.17859	0.23869	0.20394
Decision Tree classifier	8	0.12479	0.17206	0.14639
Random Forest classifier	7	0.20633	0.10276	0.13468
K-Nearest Neighbors classifier	6	0.14916	0.11970	0.13250
XGBoost classifier	7	0.21741	0.04865	0.07883
Logistic Regression	2	0.28573	0.03608	0.06329
LightGBM classifier	1	0.26814	0.02933	0.05221
Linear Support Vector classifier	2	0.30382	0.02303	0.04244
Gradient Boosting classifier	2	0.21819	0.00934	0.01790
AdaBoost classifier	5	0.16817	0.00555	0.01057

Table 6.6 presents the best performance per classifier in the Code2Vec-based analysis, sorted by the highest F1 score in descending order. The Gaussian Naïve Bayes classifier achieved the highest F1 score of 0.20394 at $k = 8$, followed by the Decision Tree classifier with an F1 score of 0.14639 at $k = 8$, and the Random Forest classifier with an F1 score of 0.13468 at $k = 7$.

Conversely, the AdaBoost classifier achieved the lowest F1 score of 0.01057 at $k = 5$, followed by the Gradient Boosting classifier with an F1 score of 0.01790 at $k = 2$, and the Linear Support Vector classifier with an F1 score of 0.04244 at $k = 2$.

6.4.2.5 Token-Based Metrics Combination Analysis

Table 6.7 presents the best metrics combination per classifier in the token-based analysis, categorised into hit-independent and hit-dependent metrics. The table shows that hit-dependent metrics are more prevalent in the best metrics combinations, consistent with the previous chapters.

The table also highlights the relative importance of each metric by indicating its frequency of appearance in the best metrics combinations. For example, the NDTT metric is part of the best metrics combination for all classifiers except the

Table 6.7: Token-Based Best Metrics Combination Per Classifier

Metric	**Classifiers**									
	AB	***DT***	***GNB***	***GB***	***KN***	***LGBM***	***LSVC***	***LG***	***RF***	***XGB***
Hit-Independent										
NTT	✓	✓		✓	✓	✓				✓
NDTT	✓	✓	✓	✓		✓	✓	✓	✓	
TRU	✓		✓			✓		✓		
Hit-Dependent										
NHS	✓							✓	✓	
NDHS								✓	✓	
NVT	✓				✓				✓	
NDVT			✓		✓	✓			✓	
TVSSR	✓	✓				✓			✓	✓
SHTSR	✓	✓	✓		✓	✓		✓		✓
SHVSR									✓	✓
NUSM		✓			✓				✓	
SMR			✓			✓			✓	✓

K-Nearest Neighbors and XGB classifiers. In contrast, the NDHS metric appears only twice in the best metrics combinations, specifically for the Logistic Regression and Random Forest classifiers.

Thus, regarding the token-based component of the second objective, the optimal combination of software metrics for vulnerability prediction includes those used by the best-performing classifier, the Gaussian Naïve Bayes classifier. These metrics are NDTT, TRU, NDVT, SHTSR, and SMR.

6.4.2.6 Code2Vec-Based Metrics Combination Analysis

Table 6.8 presents the best metrics combination per classifier in the Code2Vec-based analysis, categorised into hit-independent and hit-dependent metrics. As expected, hit-dependent metrics are more prevalent in the best metrics combinations.

The table also highlights the relative importance of each metric by indicating the frequency with which it appears in the best metrics combinations. The NDTP metric appears in the best metrics combinations for all classifiers, while the NDVP metric appears only once in the best metrics combination for the Random Forest classifier.

Table 6.8: Code2Vec-Based Best Metrics Combination Per Classifier

Metric	**Classifiers**									
	AB	***DT***	***GNB***	***GB***	***KN***	***LGBM***	***LSVC***	***LG***	***RF***	***XGB***
Hit-Independent										
NTP		✓	✓	✓					✓	
NDTP	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PRU		✓	✓		✓				✓	✓
Hit-Dependent										
NHP	✓	✓	✓						✓	✓
NDHP	✓	✓	✓		✓					✓
NVP										
NDVP									✓	
TVPSR		✓			✓					✓
PHTSR			✓		✓		✓	✓		✓
PHVSR	✓	✓								
NUPM	✓	✓	✓		✓				✓	✓
PMR			✓						✓	

Therefore, regarding the Code2Vec-based component of the second objective, the optimal combination of software metrics for vulnerability prediction includes those used by the best-performing classifier, the Gaussian Naïve Bayes classifier. These metrics are NTP, NDTP, PRU, NHP, NDHP, PHTSR, NUPM, and PMR.

6.4.3 Objective 3 Results

Evaluate the impact of hyperparameter tuning on the performance of the best-performing classifier for token-based and Code2Vec-based analyses.

This objective evaluated the impact of hyperparameter tuning on the performance of the best-performing classifier for vulnerability prediction in both the token-based and Code2Vec-based analyses.

6.4.3.1 Token-Based Parameter Grid and Best Hyperparameter Values

Table 6.9 presents the parameter grid (default sci-kit-learn values marked with an asterisk ‘*’) and best hyperparameter values for the Gaussian Naïve Bayes classifier in the token-based analysis.

Table 6.9: Parameter Grid and Best Hyperparameter Values (Token-Based Analysis)

Parameter Grid		
Hyperparameter	***Values***	***Best Value***
priors	*None**, [0.1,0.9], [0.2,0.8], [0.3,0.7], ↪ [0.5,0.5], [0.6,0.4], [0.7,0.3], [0.8,0.2], ↪ [0.9,0.1]	[0.8,0.2]
var_smoothing	$1e^{-9}$*, $1e^{-8}$, $1e^{-7}$, $1e^{-6}$, $1e^{-5}$	$1e^{-5}$

The Gaussian Naïve Bayes classifier has fewer hyperparameters to tune compared to classifiers like the Random Forest classifier. Its simplicity is its main strength, so extensive hyperparameter tuning is unnecessary.

The table shows that the best hyperparameter values for the Gaussian Naïve Bayes classifier are $priors = [0.8, 0.2]$ and $var\_smoothing = 1e^{-5}$. The 'priors' hyperparameter represents the prior probabilities of the classes, which is ideal for imbalanced datasets [Thölke et al., 2023], while 'var_smoothing' is the portion of the largest variance of all features added to variances for calculation stability [Sari et al., 2021].

6.4.3.2 Hyperparameter Tuning Results for Token-Based Analysis

Table 6.10: Pre-and-Post-Hyperparameter Tuning Results for Gaussian Naïve Bayes Classifier (Token-Based Analysis)

Metric	**Before**	**After**	$\Delta\%$
Precision	0.16780	0.15115	-9.9
Recall	0.29921	0.38016	27.0
F1 score	0.21477	0.21616	0.65

Table 6.10 presents the pre- and post-hyperparameter tuning results for the Gaussian Naïve Bayes classifier in the token-based analysis. The table shows that hyperparameter tuning improved the classifier's performance, with a 9.9% decrease in precision, a 27.0% increase in recall, and a 0.65% increase in the F1 score.

Table 6.11: Parameter Grid and Best Hyperparameter Values (Code2Vec-Based Analysis)

Parameter Grid		
Hyperparameter	***Values***	***Best Value***
priors	*None**, [0.1,0.9], [0.2,0.8], [0.3,0.7], ↪ [0.5,0.5], [0.6,0.4], [0.7,0.3], [0.8,0.2], ↪ [0.9,0.1]	[0.8,0.2]
var_smoothing	$1e^{-9}$*, $1e^{-8}$, $1e^{-7}$, $1e^{-6}$, $1e^{-5}$	$1e^{-9}$

6.4.3.3 Code2Vec-Based Parameter Grid and Best Hyperparameter Values

Table 6.11 presents the parameter grid (default sci-kit-learn values marked with an asterisk '*') and best hyperparameter values for the Gaussian Naïve Bayes classifier in the Code2Vec-based analysis.

We used the same hyperparameters as in the token-based analysis. The table shows that the best hyperparameter value for 'priors' is the same as in the token-based analysis. However, the best hyperparameter value for 'var_smoothing' is $1e^{-9}$, which differs from the value used in the token-based analysis.

6.4.3.4 Hyperparameter Tuning Results for Code2Vec-Based Analysis

Table 6.12: Pre-and-Post-Hyperparameter Tuning Results for Gaussian Naïve Bayes Classifier (Code2Vec-Based Analysis)

Metric	**Before**	**After**	**Δ%**
Precision	0.17859	0.16021	-10.29
Recall	0.23869	0.30363	27.20
F1 score	0.20394	0.20932	2.64

Table 6.12 presents the pre- and post-hyperparameter tuning results for the Gaussian Naïve Bayes classifier in the Code2Vec-based analysis. The table shows that hyperparameter tuning improved the classifier's performance, resulting in a 10.29% decrease in precision, a 27.20% increase in recall, and a 2.64% increase in the F1 score.

Therefore, in addressing the third objective, hyperparameter tuning had a mixed but net positive impact on the performance of the best-performing classifiers for both the token-based and Code2Vec-based analyses, specifically the Gaussian Naïve Bayes classifier. However, it is worth noting that the overall post-hyperparameter tuning results for the mixed-project task were lower than those for the within-project vulnerability prediction task and, thus, not actionable.

6.4.4 Objective 4 Results

Investigate the impact of dataset variability on vulnerability prediction model performance.

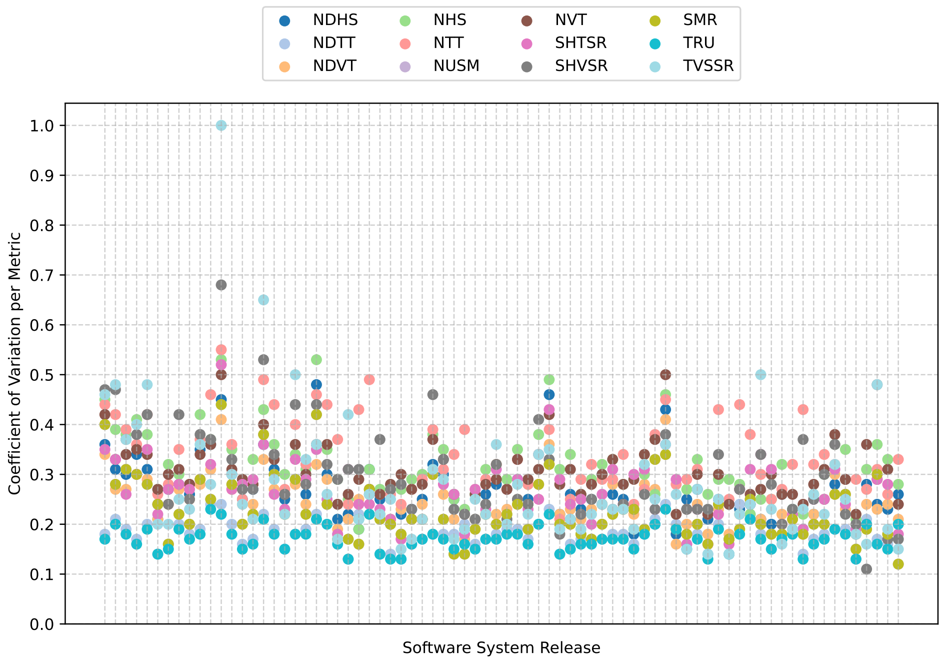
This objective investigated the impact of dataset variability on vulnerability prediction model performance by examining and comparing the Coefficient of Variation of the token-based and Code2Vec-based metrics.

Figure 6.10 shows plots of the Coefficient of Variation of the token-based metrics both for our single software system and multiple software systems.

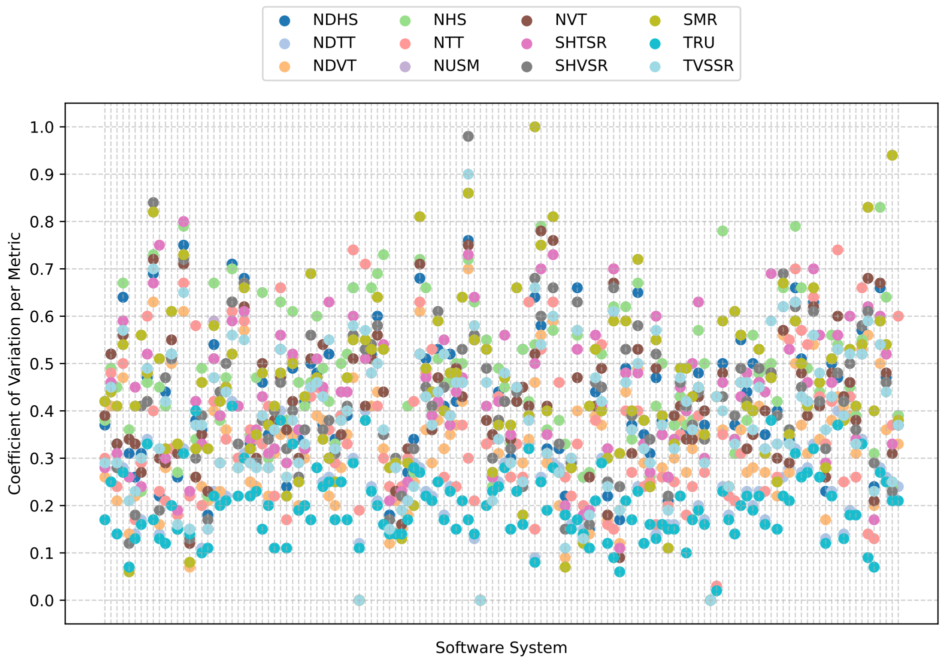
Subfigure 6.10a presents a plot of the Coefficient of Variation of the token-based metrics for the single software system. The y-axis represents the Coefficient of Variation, normalised to the range [0, 1], while the x-axis represents the software system releases used in Chapters 4 and 5. Each vertical line represents a single software system release, and the twelve data points on each line represent the Coefficient of Variation of the twelve token-based metrics for that release.

As reported in Table 4.9, the single software system dataset, Apache Tomcat (version 7), comprises 76 releases, resulting in 76 vertical lines in the plot. Out of the sixteen token-based metrics developed (Section 4.3), we employed twelve metrics in this chapter (excluding the churn-related metrics), as stated in Subsection 6.3.5, resulting in twelve data points on each vertical line.

Subfigure 6.10b shows the Coefficient of Variation of the token-based metrics for multiple software systems. The y-axis represents the Coefficient of Variation, normalised to the range [0, 1], while the x-axis represents the software systems used in this chapter's token-based analysis. Each vertical line represents a single



(a) Token-Based Metrics: Single Software System



(b) Token-Based Metrics: Multiple Software Systems

Figure 6.10: Coefficient of Variation of Token-Based Metrics

software system, and the twelve data points on each line represent the Coefficient of Variation of the twelve token-based metrics for that system.

Table 6.1 reports that the multiple software systems dataset comprises 132 software systems, resulting in 132 vertical lines in the plot. Again, out of the sixteen token-based metrics developed (Section 4.3), we employed twelve metrics in this chapter (excluding the churn-related metrics), as stated in Subsection 6.3.5, resulting in twelve data points on each vertical line.

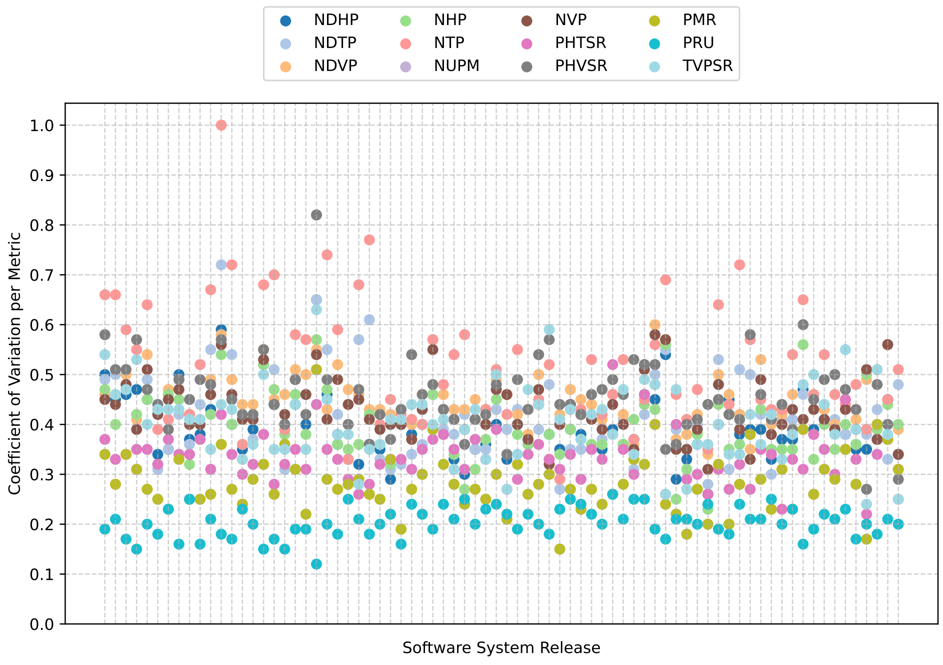
Observations from the plots reveal that the Coefficient of Variation of the token-based metrics is higher for the multiple software systems than for the single software system, as indicated by the more intense dispersion of data points in the former, particularly in terms of vertical spread. This suggests that the token-based metrics exhibit more significant variability across multiple software systems than within a single software system. This variability is expected due to differences in project characteristics, such as project size, complexity, domain, and software development practices.

Consequently, employing a dataset comprising multiple software systems, as shown in Subfigure 6.10b, as training data for vulnerability prediction models may present challenges due to the variability in the token-based metrics data across the systems.

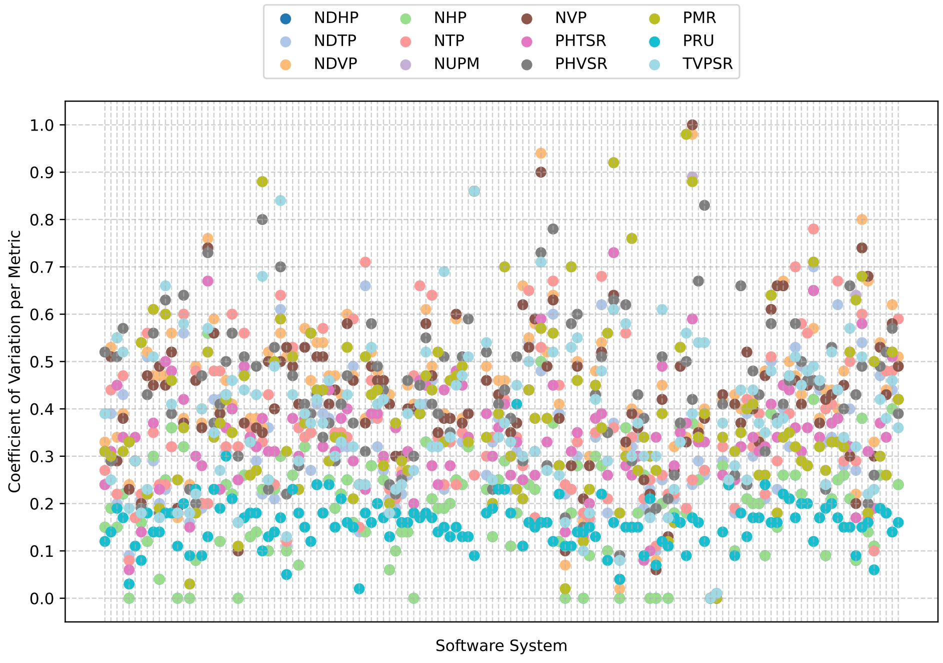
Figure 6.11 is similar to Figure 6.10, but it presents plots of the Coefficient of Variation of the Code2Vec-based metrics for our single and multiple software systems.

Subfigure 6.11a presents a plot of the Coefficient of Variation of the Code2Vec-based metrics for the single software system. The y-axis represents the Coefficient of Variation, normalised to the range [0, 1], while the x-axis represents the software system releases used in Chapters 4 and 5. Each vertical line represents a single software system release, and the twelve data points on each line represent the Coefficient of Variation of the twelve Code2Vec-based metrics for that release.

As reported in Table 4.9, the single software system dataset, Apache Tomcat (version 7), comprises 76 releases, resulting in 76 vertical lines in the plot. Out of the sixteen Code2Vec-based metrics developed (Section 5.3), we employed twelve



(a) Code2Vec-Based Metrics: Single Software System



(b) Code2Vec-Based Metrics: Multiple Software Systems

Figure 6.11: Coefficient of Variation of Code2Vec-Based Metrics

metrics in this chapter (excluding the churn-related metrics), as stated in Subsection 6.3.5, resulting in twelve data points on each vertical line.

Subfigure 6.11b shows the Coefficient of Variation of the Code2Vec-based metrics for multiple software systems. The y-axis represents the Coefficient of Variation, normalised to the range [0, 1], while the x-axis represents the software systems used in this chapter's Code2Vec-based analysis. Each vertical line represents a single software system, and the twelve data points on each line represent the Coefficient of Variation of the twelve Code2Vec-based metrics for that system.

Table 6.1 reports that the multiple software systems dataset comprises 132 software systems, resulting in 132 vertical lines in the plot. Again, out of the sixteen Code2Vec-based metrics developed (Section 5.3), we employed twelve metrics in this chapter (excluding the churn-related metrics), as stated in Subsection 6.3.5, resulting in twelve data points on each vertical line.

Similar to the observations from the token-based metrics in Figure 6.10, the Coefficient of Variation of the Code2Vec-based metrics is higher for multiple software systems than for the single software system, as indicated by the more intense dispersion of data points in the former, particularly in terms of vertical spread. Again, this suggests that the Code2Vec-based metrics exhibit more significant variability across multiple software systems than within a single one. Thus, we draw the same conclusion that employing the dataset comprising multiple software systems, as shown in Subfigure 6.11b, as training data for vulnerability prediction models may present challenges due to the variability in the Code2Vec-based metrics across the systems.

To address the fourth objective, Coefficient of Variation analyses revealed that token-based and Code2Vec-based metrics exhibit more significant variability in the dataset with multiple software systems than in the single-system dataset. This suggests that datasets with characteristics similar to the former may lead to performance issues when used as training data for vulnerability prediction models. It also implies that to achieve acceptable performance in a mixed or cross-project vulnerability prediction task, the dataset must be carefully curated to account for the variability in the software systems by ensuring that the systems are similar in terms of project size, complexity, domain, and development practices.

6.5 Discussion

In this chapter, we conducted a stress test of our information retrieval-driven vulnerability prediction techniques as a generalisability study in a mixed-project setting, focusing on both token-based and Code2Vec-based analyses. We present our findings and insights in the following subsections, discussing the performance of within-project versus mixed-project approaches, the impact of dataset variability, and the chapter's implications and recommendations.

6.5.1 Within- versus Mixed-Project Performance Comparison

For both the token-based and Code2Vec-based analyses, the within-project vulnerability prediction task yielded higher performance than the mixed-project task. The results are expected as within-project prediction is generally relatively more straightforward than mixed-project prediction due to the availability of project-specific information, such as structure, coding conventions, and development practices, which enhance model learning and accuracy. In contrast, mixed-project prediction requires models to generalise across different projects, which is challenging due to differences in project characteristics and the lack of project-specific information.

To quantify the performance difference, the within-project token-based analysis in Chapter 4 achieved a post-hyperparameter tuning precision of 0.73, recall of 0.60, and F1 score of 0.66 using the Random Forest classifier. The mixed-project token-based analysis in this chapter achieved a post-hyperparameter tuning precision of 0.15, recall of 0.38, and F1 score of 0.22 using the Gaussian Naïve Bayes classifier. Similarly, the within-project Code2Vec-based analysis in Chapter 5 achieved a post-hyperparameter tuning precision of 0.72, recall of 0.62, and F1 score of 0.67 using the Random Forest classifier. The mixed-project Code2Vec-based analysis in this chapter achieved a post-hyperparameter tuning precision of 0.16, recall of 0.30, and F1 score of 0.21 using the Gaussian Naïve Bayes classifier.

The disparities in the results highlight the challenges of generalising vulnerability prediction models across different projects and the importance of considering project-specific information when building and evaluating these models.

This discussion point extends The Significance of Vulnerable Code Patterns discussion in Chapter 4 to a macroscopic level.

6.5.2 Code Representation Sensitivity

A critical observation from Figure 6.10 and Figure 6.11 is that the disparities in dispersion between the single software system and multiple software systems are more pronounced for the token-based metrics than for the Code2Vec-based metrics (i.e., Subfigure 6.10a vs Subfigure 6.10b, and Subfigure 6.11a vs Subfigure 6.11b). This observation is important because it confirms our finding in Subsubsection 5.3.1.3 of Chapter 5 that Code2Vec is less sensitive to code changes than token-based metrics. Trivial code changes modify the token representation, which in turn affects any metrics derived from the tokens, resulting in significant variability in the metrics. In contrast, since trivial code changes do not significantly affect the AST representation, the derived metrics are less sensitive to code changes, resulting in less metric variability.

So, even though AST-based metrics, such as our Code2Vec-based metrics, may better capture the hierarchical structure and relationships within the code compared to tokens, as asserted by Liu et al. [2022a] and evidenced by our slightly improved performance in Chapter 5 over Chapter 4, the token-based metrics are more sensitive to code changes. This sensitivity is evident in the relative data homogeneity in the single software system for the token-based metrics in Subfigure 6.10a compared to the Code2Vec-based metrics in Subfigure 6.11a, in comparison to their respective multiple software systems Subfigures 6.10b and 6.11b.

6.5.3 Data-Related Challenges: A Revisit

The data issues-related observations in this chapter highlight the challenges in vulnerability prediction research, particularly in the context of cross-project vulnerability prediction, underscoring the discussion in the Background section of this chapter.

Firstly, it highlights why researchers whose methodologies we briefly highlighted in Subsection 6.2.1 needed to employ sophisticated transfer learning and domain adaptation techniques in their cross-project vulnerability prediction analyses.

Secondly, it illustrates the data-related challenges in vulnerability prediction discussed in Subsection 6.2.2, such as data generalisability, data accessibility, data preparation effort, data scarcity, label noise, and data noise.

The observations from our Coefficient of Variation analyses primarily relate to Data Noise, particularly data heterogeneity, a common problem in vulnerability prediction which arises from inconsistencies and biases in the training data and can affect model performance and generalisability. Heterogeneity in training data for vulnerability prediction can arise due to data scarcity, the lack of standardised data formats, the absence of comprehensive and consistent data sources, and the difficulty in integrating data from multiple sources. These issues complicate the problem and contribute to the development of poor-quality vulnerability prediction models.

6.5.4 Implications

The findings from this chapter have several implications for vulnerability prediction research and practice. The results highlight the difficulties and impracticalities in generalising vulnerability prediction models across different software systems. Performance disparities between within-project and mixed-project tasks, as well as metric variability across multiple systems, highlight the challenges in transferring models between projects. Data heterogeneity exacerbates these issues, introducing inconsistencies and biases that affect model performance and generalisability. Addressing these challenges requires careful consideration of dataset characteristics, including variability, quality, and representativeness. Researchers and practitioners should utilise representative datasets, employ appropriate evaluation methods, and incorporate project-specific information to mitigate these challenges.

The Coefficient of Variation analyses provide valuable insights into the variability of vulnerability prediction metrics across different systems. By quantifying metric variability, researchers and practitioners can gain a deeper understanding of the challenges posed by data heterogeneity and its implications for model development and refinement. These insights can inform the development of more robust and generalisable models by highlighting the need to address data variability, quality, and representativeness.

6.5.5 Recommendations

Researchers and practitioners should address data heterogeneity by selecting representative datasets, ensuring data quality and consistency, and considering project-specific information when developing vulnerability prediction models. Mitigating data heterogeneity can improve model performance and generalisability, making the integration of data heterogeneity analysis into the vulnerability prediction process essential.

Data-driven insights, such as the Coefficient of Variation analysis, provide valuable information on metric variability across systems and should be a staple in vulnerability prediction research. This sentiment is also echoed by Zheng et al. [2020], whose study calls for a deeper analysis of dataset attributes, given that different datasets tend to have distinct characteristics that influence results differently. Researchers and practitioners should utilise such dataset-related insights to comprehend data heterogeneity and its implications, and apply them to inform dataset selection, thereby facilitating the development of more robust and generalisable models. Furthermore, incorporating project-specific information, such as structure, coding conventions, and development practices, can also enhance model performance and generalisability.

Finally, addressing data-related challenges, such as heterogeneity, scarcity, label noise, and data noise, is difficult but crucial. Additionally, evaluating model performance for both within-project and stress testing them under mixed-project conditions is essential, as comparing performance across different tasks provides insights into generalisation challenges and opportunities for improvement.

6.6 Threats to Validity

In this section, we discuss the threats to the validity of our study, some of which are common to vulnerability prediction studies and may also apply to the previous two chapters.

6.6.1 Internal Validity

Internal validity refers to the extent to which the study's results accurately reflect the genuine relationship between variables without being influenced by external factors.

6.6.1.1 Context-Dependent Vulnerabilities

Software vulnerabilities may sometimes span multiple methods rather than being confined to one. Some methods might not exhibit vulnerability in different contexts. While it is possible to address these context-dependent vulnerabilities by considering a broader unit of analysis, this study focused on method-level vulnerability prediction.

6.6.1.2 Selection Bias

The dataset used in this chapter comprises many projects, but it represents only a minuscule fraction of the available software projects. The projects are primarily open-source and hosted on GitHub, which may not reflect the full diversity of software projects. Consequently, the results of this study may not be generalisable to all software projects.

6.6.1.3 Data Imbalance

The imbalance between vulnerable and non-vulnerable methods in the dataset could lead to biased model performance, favouring the majority class. We employed the SMOTE to balance the dataset, yet the risk of residual imbalance effects persists.

6.6.1.4 Ground Truth Estimation

Our approach to labelling methods as vulnerable or non-vulnerable in this chapter (see Subsubsection 6.3.2.2) differs from the approaches in the previous two chapters (see Subsubsection 4.4.3.4). This chapter's approach aligns with the approach used in our primary source paper, i.e., Al Debeyan et al. [2022], for the dataset used in this chapter, as noted earlier. While logically sound, we note that the approach, like other ground truth estimation approaches, are imperfect and could introduce label noise into the dataset, affecting model performance.

6.6.2 External Validity

External validity concerns how this study's findings can be generalised to other contexts outside the experimental settings.

6.6.2.1 Programming Language Generalisability

The study focused on Java-based software systems, which may limit the applicability of the findings to projects written in other programming languages. While Java is widely used, the results may not generalise to systems developed in languages with different characteristics.

6.6.2.2 Focus on Static Vulnerabilities

The vulnerabilities examined in this chapter, as well as in the preceding two chapters, are those detectable through static analysis. This study did not include those vulnerability types that manifest only during runtime, referred to as dynamic vulnerabilities. Consequently, the generalisability of the findings is confined to vulnerabilities that can be identified statically.

6.6.2.3 Limitations of Vulnerability Patterns

This study concentrated on recognised vulnerability patterns, which may not encompass emerging or unidentified vulnerabilities. Moreover, the vulnerability dataset, i.e., SARD, utilised in this research, likely does not include all known vulnerability patterns. Consequently, these models may be ineffective in identifying vulnerabilities that exhibit patterns not previously encountered in the training data.

6.6.2.4 Tool and Technique Dependence

The study utilises specific tools, including JavaParser and Apache Lucene, as well as techniques such as information retrieval and machine learning algorithms. Different tools or techniques might yield different results, and the findings may not be directly transferable if different methodologies are used.

6.7 Answer to Research Question 3

Research Question 3: How well does the information retrieval-driven software vulnerability prediction technique generalise across multiple software systems?

This chapter explored the challenges of using diverse data sources to train vulnerability prediction models, focusing on the dataset's generalisability and the impact of dataset variability on model performance. The primary objective was to evaluate the generalisability of vulnerability prediction models using both token-based and Code2Vec-based analyses as a stress test. This involved comparing within-project and mixed-project performance, analysing dataset variability, and understanding the role of hyperparameter tuning in model optimisation.

We conducted comprehensive analyses on token-based and Code2Vec-based datasets, using performance metrics such as precision, recall, and F1 score to evaluate model performance. Then, we utilised the Coefficient of Variation to assess the relative variability of metrics across our datasets. After that, we applied hyperparameter tuning to optimise model performance and compared the results between within-project and mixed-project tasks.

While performance metrics were generally poor, the findings provided valuable insights into the dataset challenges affecting vulnerability prediction analyses. The results showed significant disparities between within-project and mixed-project performance. Within-project tasks from the previous two chapters generally yielded higher performance metrics due to the availability of project-specific information and the relatively reduced data variability across the different releases of the same software system. In contrast, mixed-project tasks suffered from data heterogeneity and variability due to differences in project characteristics of different constituent software systems, resulting in lower performance metrics.

These findings have significant implications for vulnerability prediction researchers and practitioners. Firstly, the study highlights that addressing data heterogeneity and metric variability challenges is crucial for developing robust and generalisable models. Secondly, the findings indicate the need for representative datasets, appropriate evaluation methods, and normalising integrating dataset variability tests in vulnerability prediction research to facilitate the incorporation of project-specific information to mitigate the field's data-related challenges. Based on the findings, it is recommended that researchers and practitioners focus on addressing data heterogeneity by selecting representative datasets, ensuring data quality, and considering project-specific characteristics.

Future work should explore advanced techniques for improving the generalisability of vulnerability prediction models. This includes investigating methods for handling data heterogeneity, developing adaptive models to transfer knowledge across projects more effectively, and integrating additional project-specific information.

To answer the third research question of this thesis, the stress test conducted on our information retrieval-driven software vulnerability prediction technique unsurprisingly revealed the significant data-related challenges in generalising vulnerability prediction models across multiple software systems. The results highlighted the difficulties in transferring models between projects due to data heterogeneity and variability, highlighting the importance of considering project-specific information when building and evaluating these models. Regardless, the technique remains effective for within-project vulnerability prediction tasks, as evidenced by the results and findings in Chapters 4 and 5.

Chapter 7

Conclusion

This chapter concludes the thesis by summarising key findings, discussing research contributions, and outlining future research directions. The research investigated the effectiveness of information retrieval-driven techniques for predicting software vulnerabilities, utilising various machine learning models and source code representations. This concluding chapter highlights how the research objectives were met, discusses the implications of the findings, outlines the contributions to the field, and identifies potential limitations. It also suggests future research avenues that can build on this work.

7.1 Vulnerability Prediction: A Retrospective and Prospective

Software vulnerability prediction is increasingly vital as the digital landscape expands and cyber threats become more sophisticated. Vulnerabilities, i.e., security-relevant bugs in software design, development, or configuration, can be exploited by malicious actors to compromise system security, leading to financial, reputational, and even physical damages. Traditional vulnerability identification methods, such as static and dynamic analysis, are often inadequate, especially in large-scale software systems where complexity can obscure critical issues.

Machine learning techniques have shown promise in predicting software vulnerabilities in recent years. These methods utilise historical data and source code features to identify potential security vulnerabilities before they can be exploited. This thesis contributes to this emerging field by exploring information retrieval techniques to enhance vulnerability prediction through the development of security-relevant source code metrics.

The research presented here is situated within the broader context of software security. It addresses the challenges posed by increasing software system complexity and the corresponding rise in security threats. This work focuses on method-level vulnerability prediction, aiming to improve the accuracy and reliability of automated vulnerability prediction and equipping developers with tools to enhance the security of their software.

This chapter concludes the thesis by summarising key findings, discussing research contributions, and outlining future research directions. The research has investigated the effectiveness of information retrieval-driven techniques for predicting software vulnerabilities using various machine learning models and different representations of source code. This final chapter provides an overview of how the research objectives were met, the implications of the findings, contributions to the field, and limitations that may have influenced the results. It also suggests future research directions that can build on the research presented in this thesis.

7.2 Summary of Research Objectives

This research aimed to develop and evaluate an information retrieval-based approach for predicting software vulnerabilities. We accomplished this aim by pursuing the following specific objectives:

- Assess the performance of the information retrieval-based technique on a multi-release software system dataset using source code token representation.
- Assess the technique's performance using Code2Vec representation on the same dataset.
- Determine the technique's generalisation capability on a dataset comprising code artefacts from multiple software systems.

These objectives were designed to address gaps in the literature and advance current methodologies for predicting vulnerabilities in contemporary software systems, with a focus on developing security-relevant metrics and a thorough examination of data-related challenges that negatively impact the performance of vulnerability prediction models. The research developed and evaluated various models, providing valuable insights into the effectiveness of information retrieval techniques in predicting software vulnerabilities and the impact of different source code representations on model performance.

7.3 Key Findings

This section summarises the main findings of the research, aligning them with the research questions presented in Section 1.4.

7.3.1 Token-Based Prediction Performance (Within-Project)

The research demonstrated that our information retrieval-based approach performed well on a multi-release software system dataset when using token representation. The model effectively identified vulnerability-prone components by leveraging token-based features from the source code. After hyperparameter tuning, we achieved a precision of *0.73*, recall of *0.60*, and an F1 score of *0.66* using a Random Forest classifier. These results align with other studies [Ferzund et al., 2009, An and Khomh, 2015, Al Debeyan et al., 2022, Shailee et al., 2024, Moussa et al., 2022] that identified Random Forest as an effective model for vulnerability prediction.

7.3.2 Code2Vec-Based Prediction Performance (Within-Project)

Using the Code2Vec representation, the information retrieval-based technique achieved a slightly better F1 score than the token-based approach. This improvement is likely due to Code2Vec's ability to capture more contextual information through its AST-based representation [Samoaa et al., 2022]. After hyperparameter tuning, the model achieved a precision of *0.72*, recall of *0.62*, and an F1 score of *0.67* using a Random Forest classifier. This suggests that advanced source code representations, such as Code2Vec, can enhance the predictive power of vulnerability models by better capturing the semantic relationships between code elements.

7.3.3 Mixed-Project Prediction Performance

While evaluating our information retrieval-based approach in a within-project setting showed promising results, our stress test of the technique's performance across multiple software systems (mixed-project) was less consistent, an outcome we attribute to the data-related challenges discussed in Chapter 6. This highlights the challenge of generalising vulnerability prediction models across diverse systems, as system-specific characteristics can significantly impact performance. It also highlights the importance of integrating systematic approaches to dataset selection, such as Coefficient of Variation analyses, into vulnerability prediction research to help practitioners better understand dataset characteristics and make informed decisions on the suitability of a dataset for their analysis.

7.4 Contributions to the Field

This research has made several key contributions to the field of software vulnerability prediction:

- **Application of Information Retrieval Techniques**: This research pioneered information retrieval techniques for software vulnerability prediction. The approach demonstrated potential in improving the accuracy of vulnerability prediction models by providing a framework for extracting security-relevant features from source code using information retrieval methods.

- **Development of Security-Relevant Metrics**: This thesis introduced novel security-relevant software metrics derived using information retrieval techniques. These metrics enhanced the performance of vulnerability prediction models, highlighting the importance of incorporating vulnerable code patterns into machine learning models and adding valuable tools for software security analysis.

- **Evaluation of Source Code Representations**: The research evaluated various source code representations, including token-based and Code2Vec, for their effectiveness in predicting vulnerabilities, thereby contributing to the discourse on the role of semantic information in software security.

- **Generalisability of Prediction Models**: The study assessed the generalisability of vulnerability prediction models across different software systems, offering insights into the challenges and opportunities of applying these models in varied contexts and highlighting the importance of statistics-driven systematic dataset selection in vulnerability prediction research.

This thesis contributes to the broader understanding of software security by highlighting the challenges and opportunities in applying advanced machine learning techniques to vulnerability prediction. The insights from this research will be valuable to both academic researchers and industry practitioners, enabling them to develop more effective and reliable vulnerability prediction tools.

7.5 Limitations of the Study

While this research has provided valuable insights, several limitations should be noted:

- **Dataset Constraints**: The findings may be limited by the specific datasets used for model training and evaluation, including factors like software domain and dataset size.
- **Focus on the Java Programming Language**: The research experimented with software systems developed using the Java programming language, which may limit the generalisability of the findings to other languages.
- **Focus on Method-Level Vulnerability Prediction**: The research focused on method-level vulnerability prediction, which does not capture all aspects of software security.
- **Focus on Binary Classification**: The research centred on binary classification, predicting whether a method is vulnerable, without addressing multi-class classification, which could offer more detailed insights into vulnerability types.

7.6 Future Research Directions

The findings and contributions of this thesis suggest several avenues for future research. One promising direction is refining the information retrieval-driven vulnerability prediction technique. Future work could integrate additional code representations, such as graph-based models, to capture more contextual information and enhance prediction accuracy. It would also be valuable to improve the technique's generalisability across different software systems by developing advanced feature engineering methods or employing domain adaptation techniques.

Another critical area is the further exploration of Large Language Models (LLMs) in software vulnerability prediction. As LLMs advance, their potential for more accurate and context-aware vulnerability prediction increases. Our immediate future work will focus on extending our information retrieval-driven technique to leverage Retrieval-Augmented Generation (RAG) for vulnerability prediction, incorporating the latest advancements in LLMs.

7.7 Final Thoughts

Until AI-driven vulnerability prediction techniques mature, the most practical approach for achieving actionable results suitable for real-world applications is to focus on a single (multi-release) software system within a within-project setting, using a release-by-release dataset construction method and incorporating information on previously fixed vulnerabilities. This real-world implementation could be integrated into a continuous integration/continuous deployment (CI/CD) pipeline. The pipeline would automatically extract security-relevant features from each release's source code, train a vulnerability prediction model, and evaluate the model's performance. Also, since the quality of the vulnerability dataset is crucial for building accurate prediction models, such a pipeline must ensure that the vulnerability dataset used for building security-relevant metrics (features) is periodically updated to reflect the latest vulnerabilities and the vulnerability prediction model is retrained using the updated features to incorporate the latest vulnerable code patterns as new vulnerabilities are discovered.

This release-by-release approach will help mitigate several data-related challenges discussed in Chapter 6, including generalisability, accessibility, preparation effort, scarcity, label noise, and data noise. The approach will enhance data generalisability by capturing the temporal evolution of software systems, reflecting changes in code and vulnerabilities over time to inform more accurate prediction models. The structured nature of these datasets will also improve data accessibility and reduce label and data noise by ensuring more accurate labelling and cleaner data is curated throughout the software system's lifecycle. Additionally, this pre-organised format, integrated into the CI/CD pipeline, will simplify data preparation, making the training and evaluation more efficient. Finally, the approach will address data scarcity by leveraging cumulative data from multiple releases, providing a more reliable foundation for training and evaluating vulnerability prediction models over time. This described implementation methodology will help bridge the gap between academic research and industry practice, enabling developers to enhance the security of their software systems effectively and efficiently.

Despite ongoing challenges in data quality and quantity in contemporary vulnerability prediction, this work paves the way for future research, as outlined in Future Research Directions. This research has demonstrated the potential of information retrieval-driven techniques for software vulnerability prediction, highlighting their feasibility and effectiveness. It also highlights the importance of interdisciplinary approaches in software security research. Collaboration between machine learning, software engineering, and cybersecurity experts could lead to more comprehensive and robust solutions for vulnerability prediction. This could result in integrated security tools combining multiple approaches, offering developers more effective means of securing their software. Our findings contribute to the growing body of knowledge on applying machine learning and information retrieval in software security. The insights gained could lead to more robust and generalisable vulnerability prediction models, ultimately enhancing software security. This chapter concludes the thesis by summarising the contributions, acknowledging the limitations, and suggesting directions for future research. The work presented represents a significant step forward in vulnerability prediction and aims to inspire further research and innovation in this critical area, ultimately contributing to a more secure digital future.

Appendix A

Investigating the Co-Evolution of Software Bugs

This appendix features a study on the co-evolution of bug-related code artefacts in software systems, focusing on the relationship between bug-fixing and bug-inducing changes.

A.1 Introduction

Bugs significantly impact software quality and maintenance costs [Khan et al., 2020]. Their effects range from minor user inconveniences to severe issues, such as complete system crashes. Non-functional bugs [Sophia et al., 2021], which have minimal impact on functionality, often go undetected, whereas more critical bugs can cause crashes or system freezes, leading to denial of service. Security-relevant bugs, in particular, can be exploited as vulnerabilities, allowing unauthorised access to systems. A notable example is the 2003 North American blackout caused by a race condition in the General Electric Energy XA/21 monitoring software, which led to an undetected local outage [Gujral et al., 2015].

Human error is the leading cause of software bugs, with approximately 57% resulting from carelessness or oversight, leading to changes in the code that induce bugs. These errors typically become apparent during testing or post-deployment [Gujral et al., 2015].

Bug-fixing changes are deliberate efforts to resolve issues and are easily identifiable in commit histories, as developers usually note these fixes in their commit messages. In contrast, identifying bug-inducing changes is more challenging, as developers are often unaware they have introduced problematic code that could compromise their software [Nadim et al., 2020].

A.1.1 Motivation

A common misconception is that a bug's point of manifestation is the same as the origin. However, bugs may appear in one part of a software system while originating elsewhere. Thus, assuming that fixing a bug where it manifests will resolve the issue is overly simplistic and often incorrect. In reality, the changes that induce bugs can be far removed from those that fix them [Wen et al., 2019]. This disconnect undermines several traditional bug-fixing methods and the tools that rely on them, emphasising the need for more sophisticated approaches to bug management. It also highlights the importance of understanding the relationship between bug-inducing and bug-fixing changes, as well as the limitations of bug-related data, which often lacks the necessary context to trace a bug's origin and evolution.

The reductive assumption described above is the basis of the SZZ Algorithm, a popular method researchers use to identify bug-inducing changes by tracing them from bug-fixing commits [Śliwerski et al., 2005]. The algorithm traces bug-fixing commits to determine the original bug-inducing changes. Issue-tracking software often lacks details on the root cause and introduction of bugs into systems. The SZZ algorithm augments their data by offering insights into the timing of bug introductions, providing valuable information for researchers and software developers [Pokropiński et al., 2022]. However, the assumption on which the SZZ algorithm is based is inaccurate, as not all bug-fixing artefacts are directly related to the bug-inducing artefacts, which begs the question: Do the lines of code flagged as 'bug-inducing' truly represent the source of the defect?

Wen et al. [2019] explored this issue by examining the connection between bug-inducing and bug-fixing commits. They found that only 73.2% of the source code files involved in bug fixes were also involved in bug inducement, with 26.8% of bugs introduced in one source code file but resolved in another. Their study highlighted that the SZZ algorithm frequently lacks precision, as it presumes that bug fixes occur precisely at the locations where the bugs were introduced.

The ramifications of this imprecision are significant, as it threatens the validity of bug-related research, which often estimates the defect status of source code artefacts based on their bug-fixing history.

For instance, a typical bug prediction experiment broadly comprises these stages:

1. Statement of hypotheses and research questions.
2. Dataset acquisition, preprocessing and feature extraction.
3. Ground truth development.
4. Training and testing the bug prediction model.
5. Performance evaluation of the prediction result against the ground truth to answer the research questions.

The ground truth data is crucial in this process as it determines the reported model's accuracy. If the ground truth data is inaccurate, the model's predictions will be unreliable. Therefore, the accuracy of the SZZ algorithm in identifying bug-inducing changes is critical to the validity of bug prediction models.

Thus, the ideal dataset for bug prediction should comprehensively and accurately present bug-fixing details, including version control information such as the commit ID and URL, as well as associated bug-inducing details. However, information about bug-inducing changes is often missing or inaccurate because, as mentioned earlier, they are not easily trackable. The reason is that developers often inadvertently introduce them and, therefore, do not explicitly document them. Therefore, most bug prediction studies rely on bug-fixing changes to estimate the ground truth, assuming that every fixed software artefact was buggy before the fix. This approach makes it impossible to accurately capture the duration between bug introduction and bug fixing, as well as the exact location(s) of the bug-inducing changes.

A.1.2 Research Question

Inspired by Wen et al. [2019]'s work, we evaluated the SZZ algorithm's performance by investigating how closely bug-fixing artefacts co-evolve with the corresponding detached bug-inducing artefacts to deepen our understanding of software bugs. Our research addresses the following question:

To what extent do bug-fixing artefacts co-evolve with their associated detached bug-inducing artefacts?

A *detached* bug-inducing artefact refers to a file that contributed to the introduction of a bug but was not involved in its subsequent fix.

A.1.3 Research Scope

- **Programming Language:** This study exclusively utilises datasets from Java-based open-source software systems. Other programming languages are not considered.

- **Ground Truth Data:** Our ground truth data is restricted to the InduceBenchmark dataset, as used in Wen et al. [2019]'s study.
- **Unit of Analysis:** The analysis in this study is conducted at the source code file level, with no consideration of other units of analysis such as classes or methods.
- **Bug Types:** The focus is on code-related bugs within the source code. Other bug types are excluded, such as those related to design, requirements, or configuration.

A.1.4 Significance and Contribution

The SZZ algorithm is commonly used to identify bug-inducing changes, but various studies have questioned its effectiveness [Wen et al., 2019, Pascarella et al., 2018, da Costa et al., 2017, Alohaly and Takabi, 2017, Gema et al., 2020, Sophia et al., 2021, Ogino et al., 2021]. This research adds to the ongoing discussion by assessing the co-evolution of bug-fixing artefacts and their associated detached bug-inducing artefacts. Our experimental results provide a clearer understanding of the SZZ algorithm's accuracy and the reliability of the bug-inducing changes it identifies.

A.1.5 Structure of the Study

The rest of the study is organised as follows: Section A.2 provides background on code artefact co-evolution. Section A.4 outlines the study's methodology. Section A.5 presents the results. Section A.6 discusses the results and the implications of the findings. Section A.7 addresses potential threats to the study's validity. Section A.8 concludes the study.

A.2 Background

A software system's revision history reveals essential details about its evolution, indicating which components typically evolve independently and which parts change together. For example, if a database query changes whenever the associated schema is modified, we can say that the query 'co-evolves' with the schema [Zimmermann et al., 2005].

Co-evolution in software systems refers to the relationship between components that change in tandem. This relationship is often expressed using association rules: $A \Rightarrow B$, where 'A' and 'B' represent different artefacts or sets of artefacts. In this context, '$A \Rightarrow B$' implies that when 'A' changes, 'B' also changes [Zhou et al., 2019a]. Here, 'A' is the *antecedent*, and 'B' is the *consequent*.

A.2.1 Association Rule Mining

Association rule mining, a technique used in data mining, identifies relationships between variables in large datasets [Tjortjis, 2020]. Commonly applied in market basket analysis, this technique helps uncover patterns in customer purchasing behaviour [Arivazhagan et al., 2022]. For instance, it can reveal that customers who buy bread are also likely to buy butter, insights that can inform product placement strategies in supermarkets.

In software engineering, association rule mining can identify relationships between software artefacts, such as files or classes, that frequently change together. In this study, we apply association rule mining to explore the relationships between bug-fixing and bug-inducing artefacts.

We will measure the *absolute support*, *support*, and *confidence* of association rules between files modified to fix a bug and those that contributed to inducing it. These metrics are fundamental in association rule mining and will be discussed later.

A.2.2 Co-Evolution of Code Artefacts: A Hypothetical Scenario

Figure A.1 depicts a hypothetical co-evolution scenario where a junior developer modifies a database schema file (*schema.foo*) and an associated query in a Java

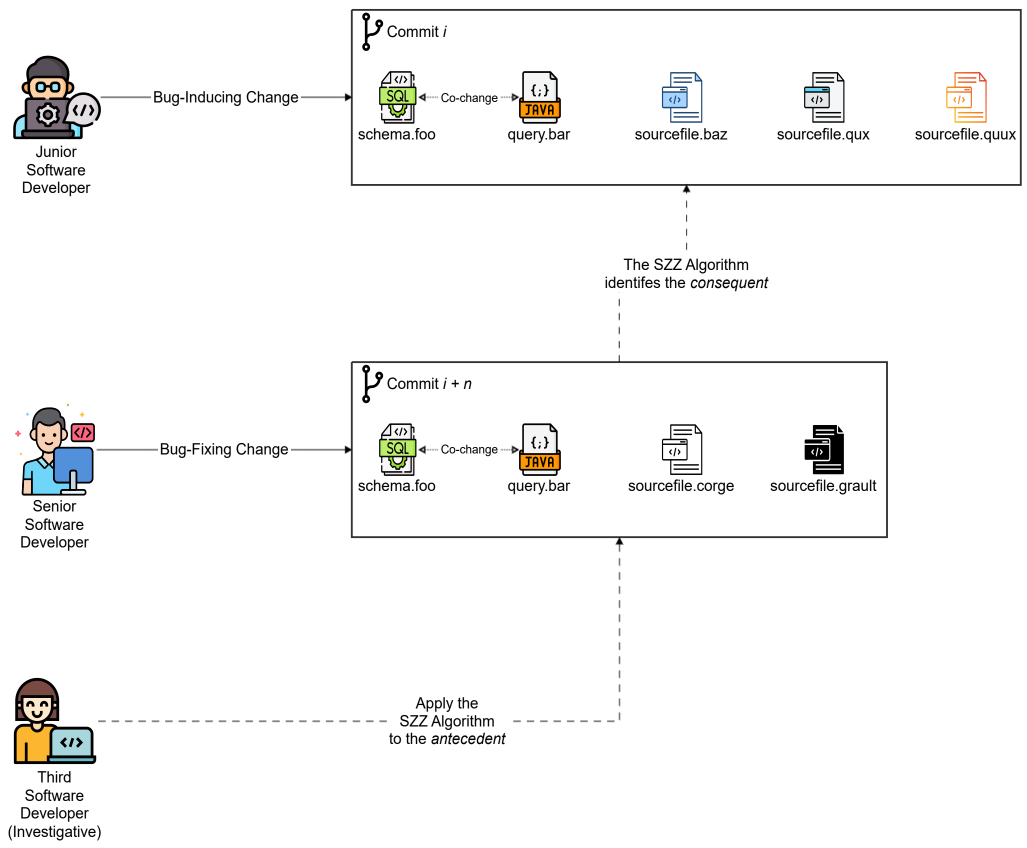


Figure A.1: Hypothetical Co-Evolution Scenario

file (*query.bar*), unintentionally introducing an SQL injection vulnerability upon committing the changes (Commit *i*).

Subsequently, a senior developer addresses the flaw by updating the schema (*schema.foo*) and refactoring the query (*query.bar*) to use Prepared Statements, committing these changes as a fix (Commit $i + n$), where n represents any number of commits between the bug introduction and its resolution.

In this scenario, *schema.foo* and *query.bar* co-evolve in both the bug-inducing and bug-fixing commits, reflecting a co-evolutionary relationship.

A third developer, investigating past bugs, might employ the SZZ algorithm to trace the original bug-inducing commit (Commit *i*) using the bug-fixing commit (Commit $i + n$).

Here, the bug-fixing commit (Commit $i + n$) serves as the antecedent, while the bug-inducing commit (Commit i) is the consequent, providing a traceable relationship for the third developer.

Files such as *sourcefile.baz*, *sourcefile.qux*, and *sourcefile.quux* represent files involved in the bug-inducing commit but not in the subsequent fix and are, therefore, termed detached bug-inducing files.

Suppose *query.bar* depends on *sourcefile.corge* and *sourcefile.grault*, yet these were not updated by the junior developer in Commit i. In that case, it can result in *sourcefile.corge* and *sourcefile.grault* remaining buggy until Commit $i + n$. This situation causes *sourcefile.corge* and *sourcefile.grault* to be buggy when *query.bar* changes, and vice versa.

Bug prediction datasets typically include only bug-fixing changes (see Subsubsection 4.4.3.4, as an example), assuming the fixed artefacts were previously buggy. However, this approach cannot represent complex scenarios, such as the one depicted in Figure A.1, where the dataset reflects only the fix at Commit $i + n$, omitting the bug-inducing changes at Commit i.

This complex scenario, akin to a '*Schrödinger's Bug*', means that the versions of *sourcefile.corge* and *sourcefile.grault* before the fix in Commit $i + n$ are simultaneously buggy and non-buggy, depending on the state of *query.bar*, thus threatening the validity of ground truth data in bug prediction studies.

This study explores this complexity to assess whether bug-fixing artefacts can predict these detached artefacts.

The goal is to evaluate whether bug-fixing files can predict bug-inducing files by analysing the strength of their co-evolutionary relationship. In this context, *bug-fixing artefacts/files* are those modified by a bug-fixing commit, while *bug-inducing artefacts/files* are those altered by a bug-inducing commit.

Framing our research within Figure A.1, we aim to determine if the co-evolution of bug-fixing artefacts—*schema.foo*, *query.bar*, *sourcefile.corge*, and *sourcefile.grault*—and their detached bug-inducing counterparts—*sourcefile.baz*, *sourcefile.qux*, and *sourcefile.quux*—is strong enough to predict the latter using the former.

A.3 Literature Review

This section reviews a few relevant literature on software artefact co-evolution, bug-inducing changes, and the SZZ algorithm, providing context for our study.

Zimmermann et al. [2005] hypothesised that association rules for code changes could predict future modifications, reveal item coupling undetectable by program analysis, and prevent errors from incomplete changes. Their study aimed to utilise co-evolving software artefacts to help developers manage related changes, akin to a recommendation system. For example, '*Developers who changed function **foo()** also changed function **bar()**.*' This approach mirrors e-commerce recommendations, such as '*Customers who bought A also bought B.*' The researchers developed a prototype tool, ROSE, which analyses code changes and predicts additional locations that may require modifications. They evaluated ROSE on eight open-source software systems, finding that its top three suggestions were correct over 70% of the time. The study concluded that ROSE is valuable for helping developers identify necessary changes and ensuring that no critical modifications are overlooked after an initial change.

Śliwerski et al. [2005] observed that developers often introduce changes that cause issues as software systems evolve. Using CVS and Bugzilla, they identified these problematic changes by linking bug reports with their corresponding fixes. They then traced the changes before the reported bug, identifying these as *fix-inducing* changes. Their analysis found 25,317 links in Eclipse, connecting 47% of fixed bugs to 29% of transactions, and 53,574 links in Mozilla, connecting 55.3% of fixed bugs to 43.91% of transactions. The study revealed that more significant changes are more likely to induce fixes, and fix-related changes are three times more likely to lead to additional fixes than simple enhancements. This work contributed to the development of the SZZ algorithm.

Wen et al. [2019] observed that despite the widespread use of the SZZ algorithm, questions remain about its accuracy in identifying 'bug-inducing' lines of code. They investigated this issue by examining bug-inducing and bug-fixing commits across 333 open-source software bugs. Their analysis revealed that the SZZ algorithm is often imprecise, as it assumes bug fixes occur at the exact locations of the bug inducements. They found that only 73.2% of the source files involved in bug fixes were also involved in bug inducement, with 26.8% of bugs introduced in one file but resolved in another. This imprecision raises concerns about the reliability of previous studies that relied on the SZZ algorithm.

Kim et al. [2006] pointed out that software researchers frequently use bug fixes to predict bugs and identify vulnerabilities within systems, yet these fixes rarely reveal the initial change that introduced the bug. They added that recognising changes that introduce bugs could uncover crucial details about the bugs' origins, such as the developers or change types most likely to introduce them. Identifying these changes, however, is challenging. The researchers developed algorithms to automatically and accurately identify bug-inducing changes with fewer errors. Their methodology employed annotation graphs to exclude non-semantic changes and outlier fixes, significantly reducing false positives and negatives. They also manually verified the accuracy of fixes to ensure reliability. The study presented improvements to the SZZ algorithm, reducing false positives by 38% to 51% and false negatives by approximately 14%. This research highlights the significance of considering bug-inducing changes in studies related to software bugs.

Linares-Vásquez et al. [2017] noted that the prevalence of mobile devices has led to numerous studies on software vulnerabilities, particularly those related to mobile applications and operating systems (OS). However, they observed that studies on OS-related vulnerabilities often cover only a small portion of known issues. The scholars investigated 660 vulnerabilities related to the Android OS to enhance their understanding in this area. They developed a taxonomy of vulnerability types within Android and applied the SZZ algorithm to identify the most susceptible layers and subsystems of the Android OS. Additionally, they assessed the lifespan of vulnerabilities by measuring the time between their introduction and resolution. The study revealed that most vulnerabilities stem from four main issues: memory buffer operations, data processing errors, inadequate access control, and insufficient input validation. It also found that third-party hardware drivers were frequently affected. Contrary to some critiques, the SZZ algorithm showed high precision in this study. The results suggest that stringent secure coding practices could mitigate many vulnerabilities, especially in data handling and memory operations.

da Costa et al. [2017] acknowledged the importance of the SZZ algorithm in bug prediction research but noted a lack of extensive evaluation of its results. They developed a framework to assess various SZZ implementations, aiming to bridge this gap, and applied it to data from ten open-source projects. The evaluation focused on three aspects: the timing of bug appearance, the future impact of changes, and the realism of bug introductions. The findings indicated that enhancements to the SZZ algorithm often overestimated the number of correctly identified bug-inducing changes. The study also found that a single bug-inducing change could precipitate hundreds of future bugs. Furthermore, at least 46% of bugs identified by SZZ implementations were traced back to changes made years earlier. The research concluded that existing SZZ implementations lacked the precision to accurately pinpoint bug-introducing changes.

Alohaly and Takabi [2017] highlighted that Version Control Systems (VCSs) are crucial in modern software development, adding that they help developers manage code versions and assist software security experts in identifying patterns of vulnerability-inducing changes. The researchers investigated whether the concept of change classification, commonly used in bug detection, could be applied to vulnerability detection. They used semi-supervised learning and text-mining techniques on their dataset. While they did not use the SZZ algorithm, they pointed out its limitations with an example where a vulnerability-inducing change and its associated fix occurred six years apart, noting that the vulnerability migrated due to file renaming. Their experiments achieved a recall between 0.6 and 0.8 and a precision from 0.63 to 1.0, demonstrating the potential of using change classification for proactive vulnerability detection.

Gema et al. [2020] challenged the common assumption in bug prediction that lines of code modified to fix a bug are the same ones that introduced it. They noted that external factors, such as API changes, can also introduce bugs, complicating the traceability of bug origins due to a lack of empirical evidence. To refine this understanding, they developed a model to identify the first software system snapshot showing buggy behaviour. They created a dataset of bug-introducing changes unrelated to source code modifications. Using this dataset, they evaluated four SZZ algorithm implementations, finding significant inaccuracies, particularly in scenarios with multiple commits. The F1 scores varied from 0.44 to 0.77, with a maximum true positive rate of 0.63. Their findings suggest that the assumption about bug origination is overly simplistic, indicating a need for more nuanced research into bug origins to enhance software development processes.

Sophia et al. [2021] investigated the effectiveness of the SZZ algorithm in identifying non-functional bug-inducing changes, focusing on aspects such as performance and security rather than direct software functionality. They observed that fixes for non-functional bugs typically occur in locations not directly linked to their induction points, rendering the SZZ algorithm less effective. The study noted that this limitation had not been widely acknowledged in previous research. In their study, the researchers analysed the accuracy of the SZZ algorithm using the NF-Bugs dataset, specifically for non-functional bugs. Their evaluation revealed that 297 out of 376 SZZ-identified bug-inducing commits were false positives, demonstrating the algorithm's ineffectiveness. Their findings highlight the need to enhance the SZZ algorithm to better address non-functional bugs.

Ogino et al. [2021] emphasised the importance of effective bug prediction techniques to improve cost efficiency in quality assurance. They critiqued current bug prediction research for not meeting essential criteria: accurate model performance evaluation, granularity to reduce manual effort and costs, and a reliable dependent variable indicating bug presence in software components. Their study aimed to evaluate and improve bug prediction models under realistic conditions. They developed their dataset by utilising eight Java projects with multiple releases, identifying fixed bugs and using an SZZ-based algorithm to pinpoint bug-inducing commits. The study highlighted the limitations of this approach in its Threats to Validity section, questioning the accuracy of the algorithm. They emphasised three critical elements for realistic settings: method-level granularity, a release-by-release dataset approach, and the bug-inducing commit as the dependent variable. However, the resulting F-Measure of 0.19 for the bug prediction model highlighted the ongoing challenge of developing effective models in such conditions.

The literature review reveals a near consensus on the SZZ algorithm's limitations in accurately identifying bug-inducing changes. These limitations have significant implications for bug prediction research as they can lead to unreliable ground truth data. This study addresses these limitations by evaluating the co-evolution of bug-fixing and bug-inducing artefacts to determine the SZZ algorithm's accuracy in identifying bug origins. We will detail our approach to this evaluation in the following Methodology section.

A.4 Methodology

Our methodology focused on identifying source code file pairs involved in both bug-fixing and bug-inducing changes and analysing their co-evolution to assess whether bug-fixing files can reliably predict detached bug-inducing files. We utilised the InduceBenchmark dataset from Wen et al. [2019]'s study[1] as our ground truth for this study.

A.4.1 Overview of the Methodology

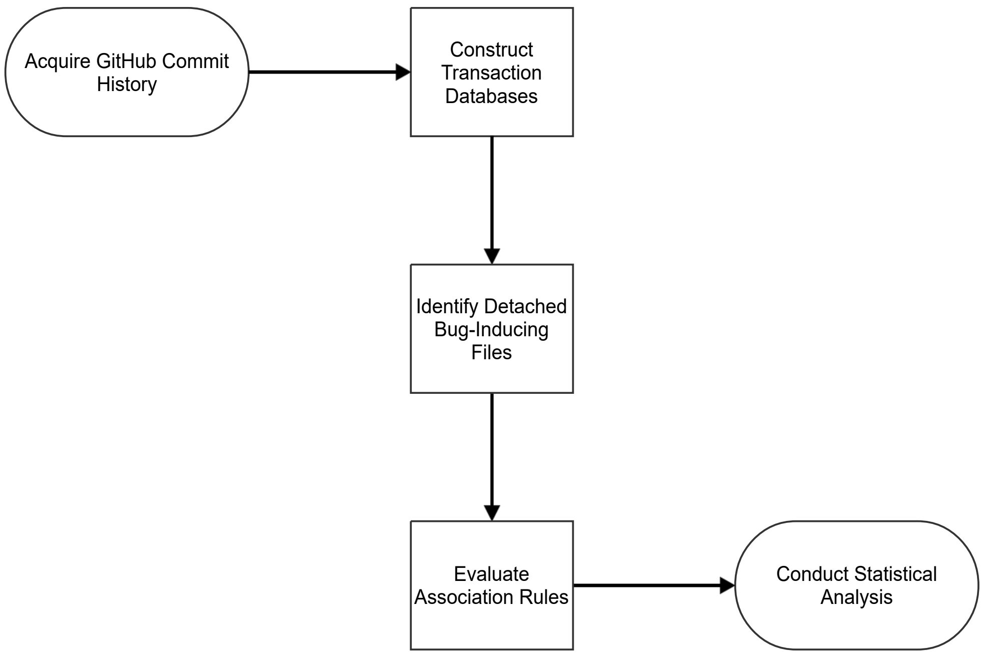

Acquire GitHub Commit History
Construct Transaction Databases
Identify Detached Bug-Inducing Files
Evaluate Association Rules
Conduct Statistical Analysis

Figure A.2: Methodology Overview

Figure A.2 outlines our methodology, divided into two main phases. Below is a summary of each phase, We present detailed descriptions in the following subsections.

1. **Phase I: Transaction Database Construction**

   I **Extract GitHub Commit History:** We used PyDriller[2] to extract the commit history for each software system.

[1]https://github.com/justinwm/InduceBenchmark
[2]https://github.com/ishepard/pydriller

II **Build Transaction Databases:** We assigned a unique ID to each modified source code file in the commit history to construct a transaction database for each software system.

2. **Phase II: Measuring Bug Co-Evolution**

I **Identify Detached Bug-Inducing Files:** We identified detached bug-inducing files by comparing bug-inducing and bug-fixing files using data from the InduceBenchmark dataset and the constructed transaction databases.

II **Evaluate Association Rules:** We assessed the association rules between bug-fixing and bug-inducing files within the transaction databases for each *bug resolution record*. A bug resolution record comprises a set of bug-fixing files and a set of bug-inducing files.

III **Perform Statistical Analysis:** Descriptive statistics were conducted to quantify the co-evolution between bug-fixing and bug-inducing files.

A.4.2 Dataset

The InduceBenchmark dataset, referenced earlier, relates to the Apache software systems studied in Wen et al. [2019]. We selected five systems from this dataset: Accumulo[3], Ambari[4], Hadoop[5], Lucene[6], and Oozie[7]. A manually curated set of bug resolution records has been compiled for these systems.

The InduceBenchmark dataset includes pairs of bug-fixing and bug-inducing commits, referred to as bug resolution records. In a bug resolution record, the antecedent is the set A_{BF}, which consists of files modified by a bug-fixing commit. The consequent is the set C_{BI}, consisting of files modified by a bug-inducing commit.

Table A.1 summarises the number of bug resolution records (abbreviated as BRRs) analysed for each software system. It also details the number of bug reso-

[3]`https://github.com/apache/accumulo`
[4]`https://github.com/apache/ambari`
[5]`https://github.com/apache/hadoop`
[6]`https://github.com/apache/lucene`
[7]`https://github.com/apache/oozie`

Table A.1: Bug Resolution Records Details

Project	**No. of BRRs**	**No. of BRRs with at least one DBIF**
Accumulo	33	29
Ambari	31	30
Hadoop	51	39
Lucene	20	19
Oozie	44	37
	179	154

lution records that include at least one detached bug-inducing file (abbreviated as DBIF). Detached bug-inducing files are crucial to our analysis. These files contributed to inducing a bug but were not involved in the subsequent fix. Further details on these files are provided later in this section.

The data in Table A.1 suggests two main types of bugs based on the complexity of their inducement. These are classified as *Type-I* and *Type-II* bugs.

Type-I bugs originate entirely from artefacts modified to fix the bug. These bugs do not have detached bug-inducing artefacts, allowing developers to resolve them by addressing the identified bug-inducing artefacts.

In contrast, *Type-II* bugs have more complex origins involving one or more detached bug-inducing artefacts. Most bugs in the bug resolution records fall into the Type-II category. Of 179 analysed bugs, 154 (86%) are classified as Type-II.

Table A.2: Dataset Details

Project	**Commits**	**Transactions**	**Modified Files**
Accumulo	11,198	7,721	77,121
Ambari	24,590	24,089	199,457
Hadoop	25,660	25,156	184,360
Lucene	35,778	34,777	267,173
Oozie	2,377	2,371	19,697

Table A.2 provides the total number of GitHub commits, transactions, and modified files for each software system as of March 2022. *A transaction is defined as a commit that modifies at least one file.*

A.4.3 Approach to Research Question

This study addresses the research question: *To what extent do bug-fixing artefacts co-evolve with their associated detached bug-inducing artefacts?*

To answer this research question, we quantified the co-evolution between artefacts modified to fix a bug and those identified as having induced it.

We analysed 179 bug reports from five Apache open-source software systems, including Accumulo, Ambari, Hadoop, Lucene, and Oozie. We then extracted the transaction database from the GitHub commit history of these systems for our co-evolution analysis. In association rule mining, a transaction is a set of items, and a transaction database is a collection of these transactions. Here, a transaction represents a commit that modifies at least one file in the software system, and a transaction database is the collection of such commits.

Our methodology involved two main phases. The first phase identified pairs of files that were modified by both bug-fixing and bug-inducing changes. The second phase analysed their co-evolution by evaluating the association rules between bug-fixing files and their associated detached bug-inducing files, where 'detached' indicates that the bug-inducing files were not part of the bug-fixing changes.

We calculated the Absolute Support, Support, and Confidence values for each association rule, followed by descriptive statistical analysis to evaluate and quantify the co-evolution between bug-fixing and bug-inducing files.

The following subsections provide detailed explanations of each phase.

A.4.4 Phase I: Transaction Database Construction

This phase aimed to construct a transaction database for each software system. A transaction, t, is defined as a commit that modifies at least one file, and a transaction database, T, is a collection of such transactions. Therefore, $t \in T$.

We used PyDriller to retrieve the commit history for each software system, focusing on the files modified in each commit. Each modified file in these transaction databases was then assigned a unique ID, ensuring that the transactions in a transaction database only include *file-modifying* commits.

As of the analysis time, PyDriller does not return modified files for merge commits, i.e., commits that merge changes from one branch into another; therefore, these commits are classified as non-file-modifying. This accounts for the difference between the figures in the 'Commits' and 'Transactions' columns in Table A.2, where the number of transactions is lower than the total number of commits for each software system.

A.4.5 Phase II: Measurement of Bug Co-Evolution

The second phase focused on measuring co-evolution for each bug resolution record, comprising three key steps: identifying detached bug-inducing files, evaluating association rules, and conducting statistical analysis.

A.4.5.1 Identification of Detached Bug-Inducing Files

The first step involved identifying the detached bug-inducing files, D_{BI}, for each bug resolution record. As defined earlier, these files contributed to inducing the bug but were not involved in its resolution. Therefore, they represent the set difference between C_{BI}, the set of files modified by a bug-inducing commit, and A_{BF}, the set of files modified by a bug-fixing commit, as described in Subsection A.4.2.

The detached bug-inducing files are mathematically represented as:

$$D_{\mathrm{BI}} = C_{\mathrm{BI}} \setminus A_{\mathrm{BF}}$$

A.4.5.2 Association Rule Evaluation

Given a bug resolution record, a_{bf} represents an element of its A_{BF}, and d_{bi} represents an element of its D_{BI}. Thus, $a_{\mathrm{bf}} \in A_{\mathrm{BF}}$ and $d_{\mathrm{bi}} \in D_{\mathrm{BI}}$. Here, $\{a_{\mathrm{bf}}\}$ is a singleton containing an element from A_{BF}, and $\{d_{\mathrm{bi}}\}$ is a singleton containing an element from D_{BI}. In this second step, we evaluated each $\{a_{\mathrm{bf}}\} \Rightarrow \{d_{\mathrm{bi}}\}$ association rule for every bug resolution record against the transactions in the five software systems.

For each $\{a_{\mathrm{bf}}\} \Rightarrow \{d_{\mathrm{bi}}\}$ association rule, we calculated the *absolute support*.

In association rule mining, absolute support is the frequency with which items co-occur in a transaction database. For example, if bread and butter are bought together in 10 out of 100 transactions, the absolute support for the rule $\{bread\} \Rightarrow \{butter\}$ is 10, indicating that the items co-occur 10 times. In our study, this metric represents the number of times a bug-fixing file, a_{bf}, co-changes with a detached bug-inducing file, d_{bi}, in a software system. Using this absolute support, we also computed the co-evolution frequency, or *support*, of a_{bf} with d_{bi}.

The $AbsoluteSupport(\{a_{\text{bf}}\} \Rightarrow \{d_{\text{bi}}\})$ is the count of transactions containing both a_{bf} and d_{bi} in a transaction database, i.e., the number of times a_{bf} co-evolves with d_{bi}.

$$AbsoluteSupport(\{a_{\text{bf}}\} \Rightarrow \{d_{\text{bi}}\}) = |\{t \in T : a_{\text{bf}} \in t \wedge d_{\text{bi}} \in t\}|$$

Support is relative to the total number of transactions in a transaction database. Continuing with the supermarket example, if bread and butter are bought together in 10 out of 100 transactions, the support for $\{bread\} \Rightarrow \{butter\}$ is 0.1. In our study, support indicates the co-evolution frequency of a_{bf} and d_{bi} in a transaction database.

Thus, $Support(\{a_{\text{bf}}\} \Rightarrow \{d_{\text{bi}}\})$ is the co-evolution frequency of a_{bf} and d_{bi}, calculated as the ratio of absolute support to the total number of transactions.

$$Support(\{a_{\text{bf}}\} \Rightarrow \{d_{\text{bi}}\}) = \frac{AbsoluteSupport(\{a_{\text{bf}}\} \Rightarrow \{d_{\text{bi}}\})}{|\{t \in T\}|}$$

Next, we calculated the *confidence*, which indicates the likelihood of encountering d_{bi} given a_{bf}.

In association rule mining, confidence is the ratio of absolute support to the number of transactions containing a_{bf}. For instance, if bread is purchased in 10 transactions and butter in 3, the confidence for $\{bread\} \Rightarrow \{butter\}$ is 0.3, indicating that butter is bought in 30% of transactions where bread is purchased. In our study, confidence measures the likelihood of encountering d_{bi} when a_{bf} is present.

Thus, $Confidence(\{a_{\text{bf}}\} \Rightarrow \{d_{\text{bi}}\})$ is the ratio of absolute support to the number of transactions containing a_{bf}.

$$Confidence(\{a_{\text{bf}}\} \Rightarrow \{d_{\text{bi}}\}) = \frac{AbsoluteSupport(\{a_{\text{bf}}\} \Rightarrow \{d_{\text{bi}}\})}{|\{t \in T : a_{\text{bf}} \in t\}|}$$

Finally, we identified the optimal values by selecting the association rules with the highest average support and confidence for each bug resolution record.

Table A.3: Co-evolution Details for ACCUMULO-3937

Bug Reference	$a_{\mathbf{bf}}$	$d_{\mathbf{bi}}$	**Abs. Sup.**	**Sup.**	**Conf.**
ACCUMULO-3937	8996	25253	32	0.00414	0.12261
ACCUMULO-3937	25237	25253	49	0.00635	0.16724
ACCUMULO-3937	***25258***	***25253***	***33***	***0.00427***	***0.41772***
ACCUMULO-3937	8996	27471	16	0.00207	0.0613
ACCUMULO-3937	25237	27471	27	0.0035	0.09215
ACCUMULO-3937	***25258***	***27471***	***14***	***0.00181***	***0.17722***
ACCUMULO-3937	8996	33387	3	0.00039	0.01149
ACCUMULO-3937	25237	33387	6	0.00078	0.02048
ACCUMULO-3937	***25258***	***33387***	***6***	***0.00078***	***0.07595***

For example, a bug resolution record, *ACCUMULO-3937*, includes bug-fixing files *{8996, 25237, 25258}* and detached bug-inducing files *{25253, 27471, 33387}*. Table A.3 shows all $\{a_{\text{bf}}\} \Rightarrow \{d_{\text{bi}}\}$ association rules between these sets. The table lists each rule's absolute support, support, and confidence values. Rows 3, 6, and 9 represent the highest average values for support and confidence, which we refer to as the record's optimal pairs.

A.4.5.3 Evaluation

In the final step, we performed descriptive statistical analysis. We calculated the five-number summary for the optimal pairs identified in Subsubsection A.4.5.2, and created boxplots for each software system. These boxplots visually depict the distribution of the optimal pairs' support, confidence, and absolute support values.

This analysis demonstrated the co-evolution of bug-fixing files with their associated detached bug-inducing files, offering a quantitative basis for benchmarking the software systems.

A.5 Results

This section presents the results of our study, focusing on the co-evolution between bug-fixing files and their associated detached bug-inducing files. We assessed the association rules linking these files and performed statistical analysis to quantify the extent of their co-evolution.

A.5.1 Absolute Support

Table A.4: Absolute Support

	Accumulo	*Ambari*	*Hadoop*	*Lucene*	*Oozie*
Min.	0	0	0	0	1
25%	1	1	1	2	2
Median	2	2	2	3	3
75%	5	4	5	6	7
Max.	51	187	79	112	203
Average	4.3	3.7	4.9	5.1	7
SD	6.6	8.8	9.5	7.1	18.6

Table A.4 summarises the Absolute Support values for the five software systems, including the minimum, 25th percentile, median, 75th percentile, maximum, average, and standard deviation for each system. The average Absolute Support values range from 3.7 to 7, with standard deviations between 6.6 and 18.6.

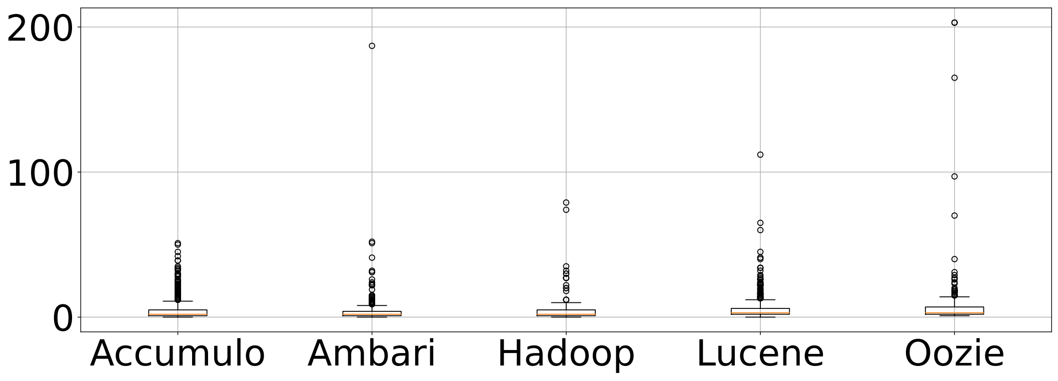


Figure A.3: Absolute Support Boxplots

Figure A.3 shows the boxplots of Absolute Support values for the five software systems based on the data in Table A.4. These boxplots provide a visual overview of the distribution of Absolute Support for each system.

A.5.2 Support

Table A.5: Support

	Accumulo	*Ambari*	*Hadoop*	*Lucene*	*Oozie*
Min.	0.00000	0.00000	0.00000	0.00000	0.00042
25%	0.00013	0.00004	0.00004	0.00006	0.00084
Median	0.00026	0.00008	0.00008	0.00009	0.00127
75%	0.00065	0.00017	0.00020	0.00017	0.00295
Max.	0.00661	0.00776	0.00314	0.00322	0.08562
Average	0.00055	0.00015	0.00020	0.00015	0.00297
SD	0.00085	0.00037	0.00038	0.00021	0.00784

Table A.5 summarises the Support values for the five software systems, including the minimum, 25th percentile, median, 75th percentile, maximum, average, and standard deviation for each system. The average Support values range from 0.00015 to 0.00297, with standard deviations between 0.00021 and 0.00784.

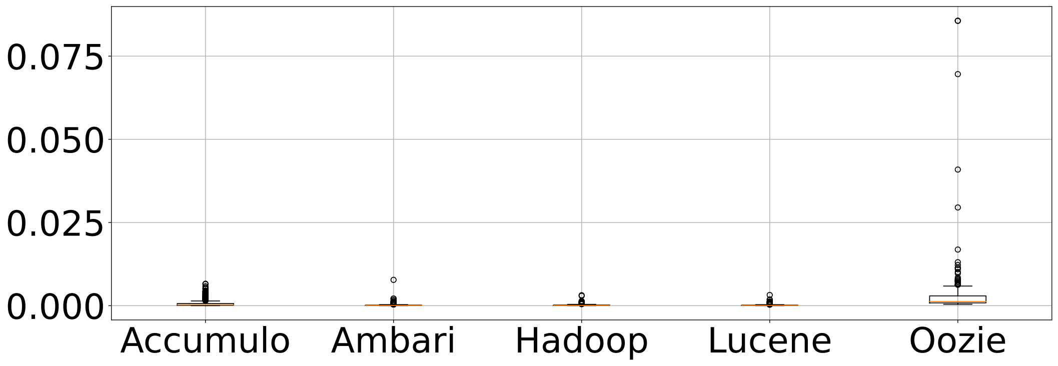


Figure A.4: Support Boxplots

Figure A.4 presents the Support boxplots for the five software systems, as derived from the data in Table A.5. These boxplots visually depict the distribution of Support values across the different systems.

A.5.3 Confidence

Table A.6 summarises the Confidence values for the five software systems, including the minimum, 25th percentile, median, 75th percentile, maximum, average, and standard deviation for each system. The average Confidence values range from 0.15615 to 0.26436, with standard deviations ranging from 0.13219 to 0.22141.

Table A.6: Confidence

	Accumulo	*Ambari*	*Hadoop*	*Lucene*	*Oozie*
Min.	0.00000	0.00000	0.00000	0.00000	0.00068
25%	0.03846	0.06667	0.03448	0.05263	0.11327
Median	0.10000	0.17742	0.09524	0.10526	0.24242
75%	0.20000	0.38503	0.25926	0.20833	0.36364
Max.	0.90909	0.85714	0.67500	0.87179	0.85714
Average	0.15615	0.24974	0.16996	0.15750	0.26436
SD	0.17060	0.22141	0.18314	0.13219	0.19982

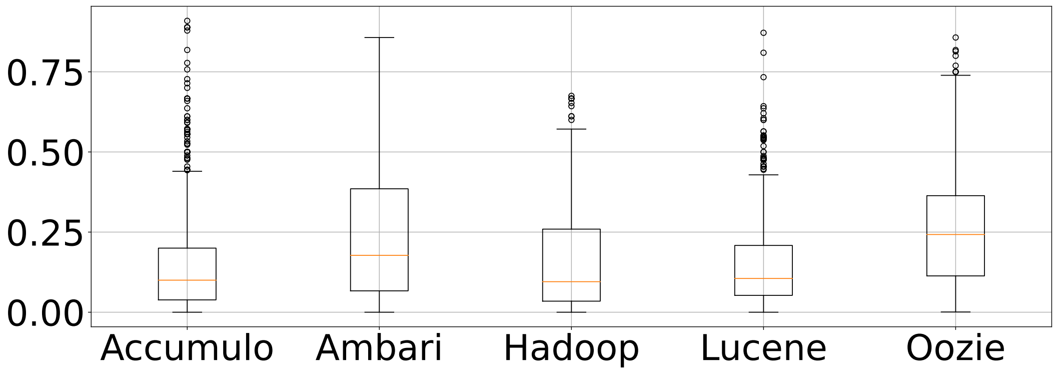


Figure A.5: Confidence Boxplots

Figure A.5 shows the Confidence boxplots for all five software systems, derived from the data in Table A.6. These boxplots illustrate the distribution of Confidence values across the different systems.

A.5.4 Co-evolving Files Per Transaction

Table A.7: Co-evolving Files Per Transaction

	Bug-Fixing Files	**DBIFs**	**Co-evolving Files/Transaction**
Accumulo	78	866	0.1222
Ambari	437	651	0.0451
Hadoop	158	193	0.0140
Lucene	58	897	0.0274
Oozie	165	483	0.2733

Table A.7 summarises each software system's co-evolving files per transaction. It lists the number of bug-fixing files, detached bug-inducing files (DBIFs), and the co-evolving files per transaction. The co-evolving file values range from 0.0140 to 0.2733.

A.5.5 Cumulative Averages for all Five Software Systems

Table A.8: Cumulative Averages for All Five Software Systems

Cumulative Average	**Value**
Number of Transactions	18822.80
Support	0.00080
Absolute Support	5
Confidence	0.19954

Table A.8 summarises the cumulative averages for transactions, support, absolute support, and confidence across all five software systems. The cumulative averages are as follows: 18,822.80 transactions, 0.00080 support, 5 for absolute support, and 0.19954 for confidence.

A.6 Discussion

Our co-evolution analysis provides valuable insights into the relationship between bug-fixing and bug-inducing files within software systems. The Absolute Support, Support, and Confidence values reflect the frequency and likelihood of co-evolution between these files. Additionally, the co-evolving files per transaction values, which capture the co-evolution rate for each software system, are essential for understanding the interaction dynamics between bug-fixing and bug-inducing files. The cumulative averages offer a broader perspective on co-evolution patterns across all five systems. In the following subsections, we discuss the results of our analysis, their implications, and their relevance to software maintenance and evolution.

A.6.1 Co-evolution Analysis

The first row in Table A.4 indicates that, except for Oozie, no bug-fixing files co-changed with detached bug-inducing files. The median absolute support across all five software systems ranges from just 2 to 3, meaning that in half of the transactions (as detailed in Table A.2), bug-fixing files co-changed with detached bug-inducing files only 2-3 times. This co-evolution trend does not significantly improve even in the third quartile, with Oozie showing the most improvement.

Overall, the metrics present poor co-evolution results, as demonstrated in Figures A.3 and A.4, and notably in Table A.8. The data shows that, on average, a bug-fixing file co-changes with a detached bug-inducing file only five times out of 18,822.80 transactions. Nevertheless, some instances of solid co-evolution were observed in individual systems.

The maximum confidence values in Table A.6 are generally very high, approaching 1, suggesting that in rare cases, detached bug-inducing files co-evolved with bug-fixing files. In such instances, co-evolution can reliably identify associated detached bug-inducing files based on the bug-fixing files. However, as shown in Figure A.5, these high confidence values, apart from those in Ambari, are often outliers, indicating their rarity.

Despite the low significance of support and confidence values, they still suggest a latent dependency between specific bug-fixing and detached bug-inducing files.

This observation aligns with Zimmermann et al. [2005], who argued that co-changes can reveal item coupling not detectable through program analysis.

In conclusion, the co-evolutionary link between bug-fixing and detached bug-inducing artefacts appears minimal. Thus, the SZZ algorithm cannot reliably use software co-evolution to infer detached bug-inducing files in bug-resolution records. These finding aligns with the previous studies by Wen et al. [2019], Pascarella et al. [2018], da Costa et al. [2017], Alohaly and Takabi [2017], Gema et al. [2020], Sophia et al. [2021] and Ogino et al. [2021] on the algorithm's limitations.

A.6.2 The Effect of Co-evolution Rate

We analysed each software system's co-evolution rate to understand Oozie's relatively higher statistical figures (see Table A.7). We calculated the co-evolution rate by summing the total number of bug-fixing files and detached bug-inducing files (DBIFs) for each system and dividing these sums by the total number of transactions, resulting in the co-evolving files per transaction.

Next, we tested the correlation between the co-evolving files per transaction values and the average support values in Table A.5. We chose the support metric for our correlation test because it is the only metric that considers all transactions within a software system. The Pearson Correlation Coefficient test produced an *r* value of *0.9618* and a *p-value* of *0.008917*, indicating a strong positive correlation between the two sets of values. This suggests that the co-evolution rate is inversely proportional to the total number of transactions in a software system. Put in another way, software systems whose co-evolution instances are very complex, i.e., involving many files, tend to have fewer transactions overall. This explains why Oozie, with fewer transactions, has higher absolute support, support, and confidence values than the other systems.

For instance, consider Ambari in Tables A.4 and A.5. Table A.5 shows a maximum support of 0.00776, corresponding to an absolute support of 187 in Table A.4. Meanwhile, Oozie's higher maximum support of 0.08562 results in an absolute support of 203. Despite the significant difference in their support values, their absolute support values are comparable (as reflected in their similar positions in Figure A.3) because the number of transactions in Oozie is significantly lower than in Ambari.

A.6.3 Bug-Contributing Artefacts

Table A.7 reveals that the total number of detached bug-inducing files consistently exceeds that of bug-fixing files across all systems. This difference arises because bug inducements often occur during feature changes or implementations, leading to commits encompassing all the code changes related to the feature. In other words, bugs are usually a side effect of feature development; thus, bug-inducing commits tend to be more extensive, often involving multiple files beyond those directly related to the bug. In contrast, bug-fixing commits tend to be more focused and involve fewer files as they target a specific bug/issue. Consequently, not every file altered in a bug-inducing commit necessarily contributed to the bug.

The term '*bug-contributing*' is similar to the '*vulnerability-contributing*' concept discussed by Meneely et al. [2013]. They argued for the use of '*vulnerability-contributing*' over terms like '*injecting*', '*fix-inducing*', or '*fault-introducing*' found in the literature. The reasoning is that the original bug-inducing commit does not necessarily prompt an immediate fix; instead, it contributes to the bug's emergence. In our context, we use '*bug-contributing artefacts*' to refer to files within the detached bug-inducing set that may not have directly caused the bug but are still considered contributors due to their involvement in the co-evolving set.

This raises the question: *how can we identify the specific bug-contributing artefacts within a set of bug-inducing files?* To explore this, we manually inspected several bug resolution records. We found that most bug-contributing artefacts within these bug-inducing files were Java source files (backend). Other artefacts included frontend files, controllers (servlets), test files, and configuration files.

Focusing solely on Java artefacts might have yielded higher co-evolution figures, but this would not provide a comprehensive analysis of software co-evolution. Software artefacts often co-evolve across layers because components from different layers typically interact to perform a function. For example, in a monolithic Java web application, changes to the backend may necessitate corresponding updates to the frontend, controllers, and test files unless a developer is simply refactoring. Therefore, a complete analysis of software co-evolution may need to consider artefacts from all layers.

Many studies, including those by Śliwerski et al. [2005] and Wen et al. [2019], have not addressed this observation or the concept of detached bug-inducing files introduced earlier.

A.6.4 Implications

Regarding Absolute Support, systems like Oozie and Ambari, which exhibit high maximum support values and significant variability, may require more targeted maintenance due to their complex and frequent co-evolution patterns. In contrast, Accumulo, with its lower and more stable values, might be easier to maintain but could also indicate less intricate interactions. The high maximum values in Ambari and Oozie suggest the presence of specific modules or components with high activity levels, potentially serving as hotspots for bugs or areas needing optimisation. Identifying these hotspots can help prioritise bug resolution and system enhancements. Oozie's minimum value of 1 and higher quartile values suggest a more interconnected system, which might benefit from strategies aimed at improving modularity to mitigate the impact of bugs.

Regarding Support, Oozie's high maximum and average support values and substantial variability indicate a more complex system with frequent and diverse interactions between bug-fixing and bug-inducing files. With lower variability, systems like Ambari and Hadoop may have more predictable interaction patterns but still require attention to outlier transactions with higher support values.

Regarding Confidence, systems with higher average and maximum confidence values, such as Oozie and Ambari, may experience more frequent and predictable interactions between bug-fixing and bug-inducing files. The variability in confidence values highlights the complexity and potential for high-impact co-evolution scenarios.

A.6.5 Recommendations

The co-evolution analysis reveals critical insights into the relationship between bug-fixing and bug-inducing files, leading to a few software maintenance and evolution recommendations.

Our findings suggest that focusing maintenance efforts on modules with high co-evolution activity can help identify and address hotspots for bugs and areas needing optimisation.

Enhancing system modularity can also help contain bugs within specific modules, improving maintainability and reducing the risk of bugs spreading across a software system.

We also deduced that prioritising transactions with higher support values can help address critical bugs and improve system reliability, as these indicate more frequent co-evolution between bug-fixing and bug-inducing files. Additionally, analysing transactions with lower variability in support and confidence can help identify stable interaction patterns and maintain system stability.

Finally, to analyse software co-evolution more comprehensively in real-world scenarios, we recommend including all evolving software artefacts, not just one file type, as in our case, Java files.

A.7 Threats to Validity

This section discusses the threats to the validity of our study, categorised into internal and external validity.

A.7.1 Internal Validity

Internal validity threats concern the study's design, execution, and analysis.

A.7.1.1 Dataset Snapshot

The dataset used in our experiment was obtained in the first quarter of 2022. Therefore, our results reflect the state of the five analysed software systems at that time. Future studies may yield different outcomes depending on the evolution of these systems.

A.7.1.2 Sample Size

Our experiment and conclusions are based on five repositories from Wen et al. [2019]'s study. Including more software systems or systems from other sources could lead to different cumulative results. However, our findings remain valid for the five systems analysed.

A.7.2 External Validity

External validity threats concern the generalisability of our results to other contexts.

A.7.2.1 Generalisation to Other Programming Languages

Different programming languages impose varying structures on artefact organisation, influencing project structure and module organisation. For example, Java mandates that methods (functions) be declared within classes, whereas JavaScript allows functions to be defined outside classes.

These design differences mean a JavaScript developer might spread related functions across multiple files, creating more association rules during functionality changes. In contrast, a Java developer would likely modify a single class file. Since our experimental repositories are Java-based, results may vary when applied to other programming languages.

A.8 Conclusion

This study examined the co-evolution of software artefacts modified to fix a bug with those that induced the bug. We analysed five Apache open-source software systems and identified two types of bugs.

Type-I bugs have straightforward origins, and developers can fully resolve them by modifying the artefacts that induced them. In contrast, Type-II bugs are induced by multiple artefacts, though not all require modification during the fix. We found that Type-II bugs are more prevalent in the analysed software systems.

The key finding of this study is that the co-evolution between bug-fixing and bug-inducing artefacts is minimal. Our results showed low median support and confidence values, ranging from 0.08% to 0.13% and 9.52% to 24.24%, respectively. These low figures indicate that the SZZ algorithm cannot reliably infer all bug-inducing artefacts based on co-evolution, making it more suitable for Type-I bugs. Future research to improve the SZZ algorithm should deprioritise co-evolution as a focus. Thus, we conclude that the co-evolutionary relationship between bug-fixing and detached bug-inducing artefacts is insignificant. This conclusion is consistent with the imprecision issues of the SZZ algorithm noted in previous studies.

Regarding the construction of bug prediction datasets, our findings suggest that it is not feasible to create a more comprehensive dataset that accurately represents the broader bug landscape, incorporating bug-inducing and bug-fixing information. This is primarily due to the unreliable, inaccurate, or unknown nature of information on bug-inducing commits. As previously discussed in the Subsection A.2.2, the ground truth in typical bug prediction datasets is generally estimated using only bug-fixing commit data. The underlying assumption is that the artefacts modified during the bug-fixing commits introduced the bug and were buggy before the fix. While this assumption is not always accurate, it remains the most viable approach without precise information about bug-inducing commits and will likely continue to be used in future studies. Regardless, acknowledging this approach's limitations is essential for interpreting the results of bug prediction studies.

Appendix B

Exploring Large Language Model-Based Vulnerability Prediction

This study explores the out-of-the-box effectiveness of Large Language Models (LLMs) in vulnerability prediction, with a focus on ChatGPT. It evaluates their performance in identifying vulnerabilities in code samples and compares them with our information retrieval-driven vulnerability prediction technique. The study offers insights into the utility, strengths, and potential improvements of LLMs in software security, particularly in the context of vulnerability prediction.

B.1 Introduction

Reliable and secure software applications are paramount in today's rapidly evolving digital landscape. As digitalisation expands, strengthening software against breaches and cyber-attacks becomes crucial. A software vulnerability can be likened to an unsecured door, allowing unauthorised access to sensitive data. To mitigate these risks, the software community is exploring various methods to identify and fix vulnerabilities in code bases [Akuthota et al., 2023].

Software vulnerabilities pose significant risks, including the compromise of sensitive information[1] and system failures[2]. Researchers have proposed machine learning and deep learning approaches for identifying vulnerabilities in source code [Hanif and Maffeis, 2022, Fu and Tantithamthavorn, 2022, Nguyen et al., 2022, Zhou et al., 2019b]. Previous methods often trained models from scratch, using algorithms such as Random Forest [Ferzund et al., 2009, Shailee et al., 2024, Dam et al., 2021] or smaller neural networks, such as Graph Neural Networks [Nguyen et al., 2022], or relied on medium-sized pre-trained models [Fu and Tantithamthavorn, 2022, Feng et al., 2020b].

Recent advancements in Large Pre-Trained Language Models have shown remarkable few-shot learning capabilities across various tasks [Xia and Zhang, 2023, Zhang et al., 2023d,e, Weyssow et al., 2023, Zhou et al., 2023]. "Few-shot learning" refers to a model's ability to learn from a few examples, making it ideal for tasks with limited training data. Introducing sophisticated models, such as OpenAI's Generative Pre-trained Transformers (GPT) series, has added a new dimension to this field. Specifically, the GPT series has exhibited a high level of proficiency in understanding, producing, and evaluating text, making it a potentially valuable tool for software code evaluation [Akuthota et al., 2023], including vulnerability prediction.

However, it is essential to meticulously evaluate the utility, strengths, and potential enhancements of LLMs in software security, particularly in the context of

[1]`https://www.bankinfosecurity.com/ms-exchange-flaw-causes-spike-intrdownloader-gen-trojans-a-16236`

[2]`https://docs.broadcom.com/doc/istr-07-sept-emea-en`

vulnerability prediction. The performance of LLMs on security-oriented tasks, such as vulnerability prediction, remains largely unexplored. While LLMs are being utilised in software engineering, notably in automated program repair [Xia and Zhang, 2023], their effectiveness in classification tasks and whether they can outperform contemporary machine learning and deep learning models in vulnerability prediction remains uncertain [Zhou et al., 2024].

This study investigates the out-of-the-box effectiveness of LLMs in predicting vulnerability. It also compares their performance with the results obtained in Chapters 4, 5, and 6 to understand the potential of LLMs in vulnerability prediction.

B.1.1 Motivation

Chapters 4 and 5 investigated vulnerability prediction using token-based and code-based source code representations, yielding promising results in a within-project setting. The token-based approach achieved a precision of 0.73, a recall of 0.60, and an F1 score of 0.66. The code-based approach yielded a precision of 0.72, a recall of 0.62, and an F1 score of 0.67.

Chapter 6 extended this research to a mixed-project setting as a stress test. This experiment showed poor performance due to issues with data quality and quantity, highlighting the need for a systematic approach to dataset selection in vulnerability prediction research. These issues were discussed in Subsection 6.2.2 and supported by empirical evidence in Subsection 6.4.4, using Coefficient of Variation analyses to compare dataset variability across within- and mixed-project settings.

Variability is a significant challenge in vulnerability prediction, affecting the generalisability and performance of models. High variability datasets often result in models with poor generalisability [Croft et al., 2022, Berggren et al., 2024], as observed in the mixed-project experiments in Chapter 6.

With the rise of LLMs, it is speculated that these models could help address some of the data-related challenges (see Subsection 6.2.2) in vulnerability prediction, particularly the issues related to data quantity, such as accessibility and scarcity, given that these models are trained on vast amounts of data. However, their impact on data quality remains to be assessed.

This study examines the out-of-the-box effectiveness of LLMs in vulnerability prediction and compares their performance with that of contemporary machine learning and deep learning models. Specifically, we focus on ChatGPT to evaluate its proficiency in identifying vulnerabilities within diverse code samples and its potential for classification tasks in vulnerability prediction.

B.1.2 Research Question

This study addresses the following research question:

> *How well do Large Language Models perform on method-level vulnerability prediction tasks?*

We evaluate the effectiveness of Large Language Models, particularly the popular ChatGPT, in identifying vulnerabilities within code samples. Their performance is compared to contemporary machine learning and deep learning models. The results are analysed to determine the potential of LLMs in vulnerability prediction.

B.1.3 Research Scope

The research scope for this study includes:

- **Programming Language:** The datasets are written in Java, with a focus on vulnerabilities in Java methods. Other sources, such as web services, annotations, and configuration files, are not considered.
- **Method-Level Vulnerability Prediction:** The focus is on predicting vulnerabilities at the method level rather than at the class or file level.
- **Within- and Mixed-Project Vulnerability Prediction:** The study evaluates the LLM-driven vulnerability prediction in both within- and mixed-project settings, using datasets comprising multiple releases of a single software system and multiple software systems.
- **Binary Classification:** The study uses binary classification to predict whether a method is vulnerable without considering multi-class classification (i.e., predicting the specific type of vulnerability).

B.1.4 Significance and Contributions

This study assesses the out-of-the-box effectiveness of LLMs in predicting software vulnerabilities, comparing their performance with that of contemporary machine learning and deep learning models. It provides insights into the strengths, utility, and potential enhancements of LLMs in software security, particularly in the context of vulnerability prediction. The empirical evidence supports the effectiveness of LLMs in this domain, laying a foundation for future research and suggesting new avenues for exploration.

The study also discusses the practical implications of using LLMs for vulnerability prediction and their potential impact on software security practices. It contributes to understanding LLMs in software security and expands the knowledge base.

B.1.5 Structure of the Study

Section B.2 provides background information on generative artificial intelligence, Large Language Models, and ChatGPT. Section B.3 presents a literature review on vulnerability prediction using Large Language Models. Section B.4 outlines the methodology, including dataset selection, data preprocessing, and model evaluation. Section B.5 presents the experimental results. Section B.6 presents the discussion of the findings. Section B.7 addresses threats to validity. Finally, Section B.8 concludes the study.

B.2 Background

The previous section introduced Large Language Models and briefly discussed their potential in software engineering, including their primary applications in natural language processing tasks, automated program repair, code generation, and other generation-based tasks. The section also highlighted the potential of Large Language Models in vulnerability prediction, particularly in identifying vulnerabilities within code samples.

This section provides additional background on generative artificial intelligence, Large Language Models, and ChatGPT, as well as their potential applications in software security, with a focus on vulnerability prediction.

B.2.1 Generative Artificial Intelligence and Large Language Models

Generative artificial intelligence has become a pivotal field, transforming domains such as computer vision, natural language processing, the creative arts [Cao et al., 2023], and requirements engineering [Vogelsang and Fischbach, 2024].

Generative models have a long history in artificial intelligence, dating back to the 1950s with the introduction of Hidden Markov Models (HMMs) [Knill and Young, 1997] and Gaussian Mixture Models (GMMs) [Reynolds et al., 2009]. These early models generated sequential data, such as speech and time series.

Generative AI focuses on creating algorithms and models that generate synthetic data that closely resembles real-world data. This capability has significant implications for the entertainment, healthcare, and finance industries. Applications include image synthesis, text generation, music composition, and human-like chatbots [Zhang et al., 2023a].

The advent of deep learning significantly improved the performance of generative models. The availability of large-scale datasets and advancements in deep learning techniques have driven the rapid development of Generative AI [Cao et al., 2023]. The growing interest in and impact of Generative AI are evident in recent statistics. Precedence Research reported that the global market for Generative AI was valued at USD 10.79 billion in 2022. It is projected to reach approximately USD 118.06 billion by 2032, with a Compound Annual Growth Rate (CAGR) of 27.02% from 2023 to 2032[3]. This surge in market demand highlights the recognition of Generative AI as a powerful tool with immense potential across various industries [Bandi et al., 2023].

Generative AI encompasses diverse applications, including StyleGAN and OpenAI's GPT series. StyleGAN [Karras et al., 2019], developed by NVIDIA, revolutionised image generation by producing highly realistic and varied images. It employs a style-based approach, manipulating visual attributes to enable new creative dimensions in digital art. Meanwhile, OpenAI's GPT-3 transformed natural language processing [Brown et al., 2020]. Its massive scale and transformer architecture generate human-like text with impressive fluency and coherence, excelling in tasks such as question answering, essay writing, and conversation. These examples demonstrate the potential of Generative AI to transform creative industries, content generation, and human-machine interaction, paving the way for further advancements.

B.2.2 The Generative Pre-trained Transformer Series and ChatGPT

Large Language Models are a subset of Generative AI models focused on natural language processing tasks [Yu et al., 2023]. The GPT series by OpenAI exemplifies LLMs, known for their large size, transformer architecture, and impressive performance across various tasks[4]. ChatGPT, a variant of the GPT series, is tailored for conversational tasks. It generates human-like text, engages in dialogue, and understands context. These models are designed for user interaction, answering questions, and providing information, making them ideal for chatbot applications[5]. The GPT series has played a pivotal role in advancing Generative AI, particularly in the

[3]https://www.globenewswire.com/en/news-release/2023/05/15/2668369/0/en/Generative-AI-Market-Size-to-Hit-Around-USD-118-06-Bn-By-2032.html/

[4]https://platform.openai.com/docs/models

[5]https://chat.openai.com/

field of Natural Language Processing. As of this writing in Q3, 2024, the series includes various models, such as the GPT-4o, GPT-4o-mini, GPT-4, and GPT-3.5-turbo. These models understand and generate natural language or code, with some also accepting image inputs. GPT-4o, the most advanced, generates text at an impressive speed and excels in vision and non-English language tasks. GPT-4o-mini, a smaller yet capable model, is ideal for tasks previously relying on GPT-3.5-turbo. It offers higher intelligence and multimodal capabilities at a lower cost, making it suitable for smaller vision-related tasks. GPT-4, OpenAI's current flagship series, is renowned for its advanced reasoning and broader knowledge. It outperforms the previous series in complex reasoning situations. The GPT-3.5-turbo model (from the GPT-3 series), optimised for chat but effective for non-chat tasks, remains available as of the time of writing and has been instrumental in various applications, from natural language understanding to code generation. LLMs are characterised by their vast size, extensive training data, and transformer architecture. For example, GPT-3 is trained on 175 billion parameters, while GPT-4, OpenAI's current flagship, is trained on one trillion parameters [Yu et al., 2023].

B.2.3 Large Language Model Applications in Software Vulnerability Prediction

LLMs have shown significant promise in software engineering, particularly in automated program repair, code generation, and code summarisation. Their ability to understand and generate code makes them valuable for software development and maintenance. The recent surge in Generative AI has sparked interest in applying LLMs to software security, particularly in predicting vulnerabilities in code. Recent studies have examined the effectiveness of LLMs in this domain. We contribute to this discussion by evaluating ChatGPT's performance in vulnerability prediction using OpenAI's GPT-3 and GPT-4 series models, with training data up to September 2021 and October 2023, respectively. We assess the out-of-the-box performance of these models without fine-tuning[6] or providing any specific context or prompts related to known vulnerabilities, focusing solely on method-level code samples. We

[6]`https://platform.openai.com/docs/guides/fine-tuning`

then compare these results with those obtained in Chapters 4, 5, and 6 to gauge the potential of LLMs in vulnerability prediction. Finally, we also assess the performances of the GPT-3 and GPT-4 series models to understand their advancements and potential in software vulnerability prediction tasks.

B.3 Literature Review

This section reviews studies on the application of LLMs in vulnerability detection and prediction, providing an overview of the current state-of-the-art in vulnerability detection using LLMs.

Akuthota et al. [2023] emphasised the critical need to secure software against breaches and cyber-attacks, especially in an increasingly digital world. They noted that traditional methods often fail to manage the complexity of modern software systems, prompting the exploration of advanced machine learning models. They investigated the use of Large Language Models, specifically the GPT-3.5-Turbo model, to detect and monitor software vulnerabilities. The study aimed to evaluate the effectiveness of GPT-3.5-Turbo in identifying vulnerabilities within software code, enhancing software security, and release management through continuous monitoring. Their methodology involved using the OpenAI interface to interact with GPT-3.5-Turbo, developing a function called "find security issues and generate fix" to scan code snippets, identify vulnerabilities, and suggest potential fixes. Data from documented vulnerabilities were analysed, and automated testing tools, such as the OWASP Benchmark, were utilised to streamline the evaluation process. The study achieved an accuracy of 0.77 in detecting vulnerabilities across 2,740 test cases, identifying various types of vulnerabilities, including SQL Injection, Cross-Site Scripting (XSS), and Command Injection. The results demonstrated that GPT-3.5-Turbo can effectively analyse code to predict security flaws, making it a valuable tool for preliminary code reviews. The research concluded that LLMs, such as GPT-3.5-Turbo, show significant promise in improving software security by accurately identifying vulnerabilities. However, the study also noted the need for ongoing refinement to address biases and enhance detection capabilities. It recommended that future work focus on training LLMs with code-based data, optimising prompts, and exploring different parameters to improve model performance.

In response to the increasing complexity of web applications and the rise in security vulnerabilities, Szabó and Bilicki [2023] investigated the use of LLMs, particularly GPT models, to enhance web application security. They noted that traditional vulnerability detection methods often require significant human intervention and may not fully address the complexities of modern web frameworks. The study aimed to assess the effectiveness of GPT models in detecting Improper Isolation or Compartmentalisation (CWE-653) vulnerabilities within web application source code. The goal was to automate the detection of these vulnerabilities, reducing the need for extensive manual code reviews. Their methodology involved a multi-step process using the GPT Application Programming Interface (API) for static code analysis of Angular web applications. The steps included preprocessing and minifying the source code of selected open-source Angular projects, using GPT models to identify and classify sensitive data elements, mapping the codebase into JSON structures for further analysis, assessing the protection levels of sensitive code segments based on predefined criteria, and comparing GPT-based analysis results with manual evaluations to assess accuracy. Few-shot examples and chain-of-thought prompting techniques improved the models' interpretive accuracy. The results showed that GPT-4 significantly outperformed previous models, achieving an 88.76% vulnerability detection rate. GPT-4 effectively understood the context and semantics of the source code, accurately detecting and classifying sensitive data segments. However, challenges were noted in handling highly modular code and identifying services managing multiple types of sensitive data. The study concluded that GPT-4 exhibits considerable potential for enhancing web application security through automated code inspection, thereby reducing the need for manual reviews and improving overall security. The study recommended that future work focus on refining prompts, exploring GPT models for other vulnerabilities and web frameworks, and addressing challenges like modular code handling and complex data flow detection.

Zhou et al. [2024] addressed the critical issue of software vulnerabilities, which can lead to severe consequences, including data breaches and system failures. They noted that while machine learning and deep learning models such as CodeBERT have been used for vulnerability detection, the emergence of Large Pre-Trained Language Models such as GPT-3.5 and GPT-4 offers new possibilities. The study aimed to evaluate the effectiveness of LLMs, particularly GPT-3.5 and GPT-4, in detecting software vulnerabilities and to determine whether they could outperform medium-sized models, such as CodeBERT, especially in classification tasks related to software security. Additionally, the research explored the impact of different prompt designs on the performance of these models. Using ChatGPT, based on GPT-3.5 and GPT-4, the researchers experimented with various prompt designs to enhance vulnerability detection. They utilised in-context learning (iCL) to avoid the computational cost of fine-tuning large models. Prompts included task descriptions, role descriptions, project information, and examples from the Common Weakness Enumeration (CWE) database. The performance of these prompts was compared to that of a fine-tuned version of CodeBERT, using a dataset of vulnerability-fixing commits from C/C++ software repositories. Results showed that the base prompt alone was inadequate, with GPT-3.5 achieving only 50% accuracy and predicting all samples as non-vulnerable. However, incorporating external knowledge from CWE examples and training data significantly improved performance. GPT-3.5 achieved 62.7% accuracy by combining random sampling and retrieval of similar code, surpassing CodeBERT's 60.3% accuracy. GPT-4, using the CWE examples prompt, outperformed CodeBERT by 34.8% in accuracy, highlighting its superior capability in vulnerability detection. The study concluded that LLMs, particularly GPT-3.5 and GPT-4, show considerable promise in enhancing software vulnerability detection, especially when well-crafted prompts are used. While GPT-3.5 performed competitively with CodeBERT, GPT-4 demonstrated even more tremendous potential, indicating significant progress in the field. The researchers emphasised the need for further investigation into local and specialised LLMs, improving precision and robustness, addressing the long-tailed distributions of vulnerability types, and fostering trust and synergy with developers. These findings suggest that LLMs could be crucial in future software security frameworks if optimised and tailored to specific needs.

Yıldırım et al. [2024] addressed the growing concerns around API security, noting that as APIs become integral to software development, they also introduce unique security risks. The study aimed to compare the effectiveness of static code analysers and LLMs in detecting API vulnerabilities, particularly those listed in the OWASP Top 10 API Security Risks. The primary objective was to evaluate how well these tools identify 40 API vulnerabilities in source code, each representing a category within the OWASP Top 10. The research sought to highlight the potential advantages of LLMs over traditional static code analysers. Their methodology involved evaluating ten static code analysers and four popular LLMs (ChatGPT 3.5, ChatGPT 4, LLaMA 2, and Bard) against 40 Python API code samples, each containing specific vulnerabilities aligned with the OWASP API Top 10 for 2019. The tools were assessed on accuracy in detecting vulnerabilities, providing correct Common Weakness Enumeration titles and numbers, and explaining the identified issues. The results showed significant differences in performance. ChatGPT 4 was the most effective LLM, with detection rates of 62.5% using the first prompt and 42.5% using the second. LLaMA 2 was the least effective. Static code analysers generally had lower detection rates, with Snyk leading at 25%, while tools like Pylint, Pyre, and Trivy failed to detect any vulnerabilities. The study highlighted that ChatGPT 4 demonstrated a deep understanding of complex API security issues, achieving 100% accuracy in specific OWASP categories. However, both LLMs and static code analysers showed variability across different OWASP categories, suggesting that a multi-tool approach might be necessary for comprehensive vulnerability detection. The study concluded that while static code analysers are helpful, their effectiveness in detecting API vulnerabilities is significantly lower than that of LLMs, mainly when the latter are appropriately prompted. ChatGPT 4 emerged as the most effective LLM tested, indicating its potential as a superior tool for API vulnerability detection. The researchers suggested combining multiple LLMs with static code analysers could offer a more comprehensive approach to API security. They also recommended that future research focus on improving the precision and robustness of LLMs, exploring specialised LLM solutions, and addressing privacy and security concerns related to using LLMs in vulnerability detection.

These studies highlight the growing interest in applying LLMs to software security, particularly in vulnerability detection. They demonstrate their potential in identifying vulnerabilities to enhance software security and automate the detection of vulnerabilities. The studies also highlight the importance of well-crafted prompts, external knowledge, and specialised LLMs in improving vulnerability detection and prediction accuracy, as well as GPT-4's superior performance in vulnerability prediction tasks. The following sections outline the methodology employed in this study to assess the effectiveness of LLMs, specifically ChatGPT, in predicting vulnerability.

B.4 Methodology

This section outlines our methodology for evaluating LLMs in the context of vulnerability prediction. It provides an overview of the methodology, detailing the dataset, data preprocessing, and model evaluation.

B.4.1 Overview of the Methodology

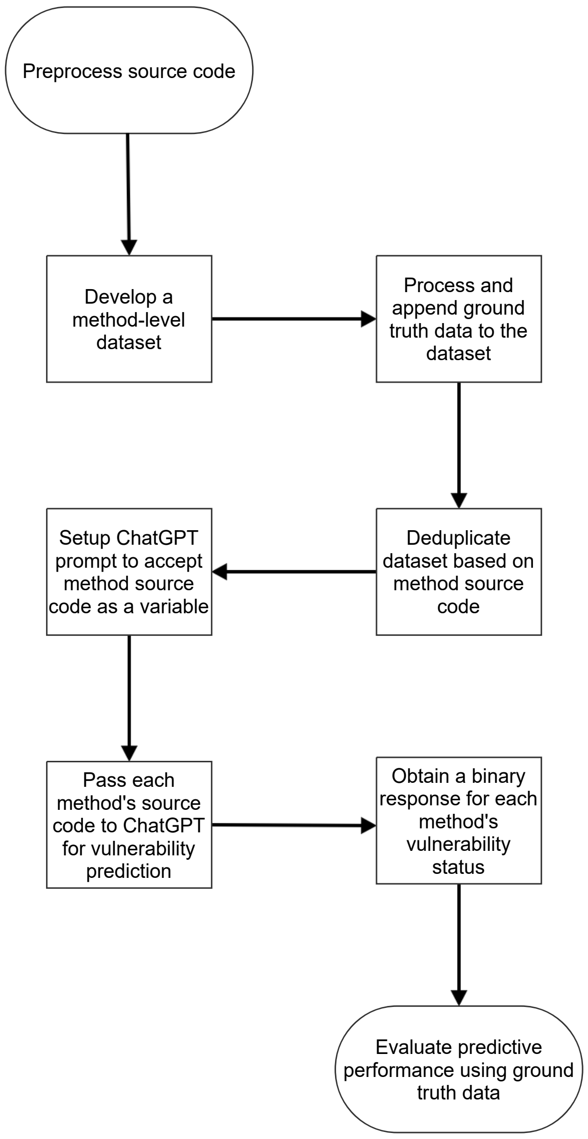

Preprocess source code
Develop a method-level dataset
Process and append ground truth data to the dataset
Deduplicate dataset based on method source code
Setup ChatGPT prompt to accept method source code as a variable
Pass each method's source code to ChatGPT for vulnerability prediction
Obtain a binary response for each method's vulnerability status
Evaluate predictive performance using ground truth data

Figure B.1: LLM-Based Vulnerability Prediction Methodology Overview

Figure B.1 illustrates our methodology, which comprises four phases: Dataset Preparation, Prompt Construction, Vulnerability Prediction, and Performance Evaluation.

The first phase, dataset preparation, includes the first four steps of the methodology (Figure B.1). It involved preparing the dataset to ensure data quality and relevance for vulnerability prediction.

1. **Source Code Preprocessing:** Preprocess source code samples to remove irrelevant information, such as comments.

2. **Method-Level Dataset Development:** Extract method-level code samples, removing unqualified code elements and methods, such as abstract and test methods.

3. **Ground Truth Annotation:** Annotate the dataset with vulnerability status labels.

4. **Data Deduplication:** Remove duplicate entries to ensure data quality.

The second phase, prompt construction, is the fifth step of the methodology (Figure B.1). It involved constructing prompts that guide LLMs in identifying vulnerabilities in code samples.

The third phase, vulnerability prediction, includes the sixth and seventh steps of the methodology (Figure B.1). It focused on predicting vulnerabilities in method-level code samples using LLMs.

1. **Method Source Code Input:** Provide method-level code samples to LLMs for vulnerability prediction.

2. **Binary Classification:** Obtain a binary response from LLMs, classifying methods as vulnerable or non-vulnerable.

The last phase, performance evaluation, is the final step of the methodology (Figure B.1). It used ground truth information to evaluate LLM performance in vulnerability prediction using precision, recall, and F1 score metrics.

The methodology provides a structured approach for analysing LLMs' effectiveness in vulnerability prediction. The following subsections detail the dataset, preprocessing, and model evaluation.

Where possible, we refer the reader to relevant sections in previous chapters, where concepts, resources, or methodological techniques have already been discussed, rather than repeating the information here to avoid redundancy.

B.4.2 Dataset

To facilitate performance comparison with our information retrieval-driven technique from previous chapters, we used the same datasets from Chapters 4 and 6: one comprising multiple releases of a single software system, i.e., within-project dataset and another comprising multiple software systems, i.e., mixed-project dataset.

The first dataset is Apache Tomcat 7, an open-source web server. It includes 76 releases from 7.0.0 to 7.0.108, each containing Java source code files. For details, see Chapter 4, specifically Subsection 4.4.2 and Subsubsections 4.4.2.1, 4.4.2.2, and 4.4.3.4. We used this dataset for within-project vulnerability prediction.

The second dataset comprises code artefacts from multiple software systems from Al Debeyan et al. [2022] and Reis and Abreu [2021]. For details, see Chapter 6, specifically Subsection 6.3.2 and Subsubsections 6.3.2.1 and 6.3.2.2. We used this dataset for mixed-project vulnerability prediction.

B.4.3 Data Processing

The data processing steps are similar to those in Chapters 4 and 6: preprocessing source code, extracting method-level details, annotating methods with vulnerability labels, and removing duplicates.

B.4.3.1 Data Preprocessing

We utilised JavaParser to parse source code files, removing irrelevant information to focus on the code's logic and structure. Irrelevant code artefacts, such as abstract and test methods, were excluded to ensure the use of only relevant data for vulnerability prediction.

B.4.3.2 Source Code Extraction

We used JavaParser to extract method-level code samples from the source code files. This process resulted in two datasets of method-level code samples: a within-project method-level dataset and a mixed-project method-level dataset.

Unlike previous chapters, which used token-based and code-based representations, this study employed raw code samples as input. We provided the code samples directly to the LLMs for vulnerability prediction without any transformation.

B.4.4 Ground Truth: Estimation

Ground truth is essential for evaluating LLM performance. We annotated the datasets with vulnerability status labels. For detailed ground truth estimation for each of the two datasets, see Chapter 4, specifically, Subsubsections 4.4.2.2 & 4.4.3.4, and Chapter 6, specifically, Subsubsections 6.3.2.2.

B.4.5 Data Deduplication

Although data leakage is not a concern in this study due to the use of pre-trained LLMs, we have deduplicated our data to ensure data quality. Removing duplicates enhances computational efficiency, storage, and performance metrics and reduces the financial costs associated with using the OpenAI API service.

B.4.6 OpenAI API

We utilised the OpenAI API[7] through HTTP requests to interact with LLMs. The API provides access to various models for text generation, classification, and other Natural Language Processing tasks. The API offers a straightforward interface for interacting with LLMs, allowing users to submit requests and receive responses. The API documentation provides comprehensive guidance on model usage, request formats, response structures, and streamlining interaction with LLMs.

B.4.7 Prompt Construction

Prompt construction is crucial for guiding the responses of LLMs. We carefully constructed prompts to direct the models to predict vulnerabilities based purely on code logic and structure, excluding external context.

We employed two prompts[8]: system prompts and user prompts. The system prompt instructed the LLMs to assess Java methods for vulnerabilities based on their

[7] `https://platform.openai.com/docs/api-reference`
[8] `https://platform.openai.com/docs/guides/prompt-engineering`

source code. The user prompt presented the code, guiding the models in classifying potential vulnerabilities in a binary manner through logical analysis.

You are an advanced vulnerability prediction system that analyses Java method source code. Your task is to predict whether a given Java method contains a security vulnerability based on its source code.
When provided with the source code of a Java method, analyse it thoroughly and determine if it has any security vulnerabilities. Your response should be “1” if a vulnerability is present and “0” if no vulnerability is detected.
Make sure to base your predictions solely on the code provided. Do not consider any external factors or additional context.

Figure B.2: System Prompt for Vulnerability Prediction

Figure B.2 shows the system prompt for vulnerability prediction. It directs the LLMs to assess Java method source code, focusing exclusively on code logic and structure without considering any external context.

Analyse the following Java method source code for any security vulnerabilities. Return your analysis in a JSON object that follows this schema: {"prediction": PREDICTED_VALUE}.
Assign "1" to the prediction property if the method contains a security vulnerability; otherwise, assign "0" to the prediction property.
Method Source Code:[METHOD_SOURCE_CODE]

Figure B.3: User Prompt for Vulnerability Prediction

Figure B.3 presents the user prompt for vulnerability prediction. It directs the LLM to analyse Java method source code for security vulnerabilities and return a binary classification. The prompt includes two placeholders: ‘[PREDICTED_VALUE]’ for the predicted vulnerability status and ‘[METHOD_SOURCE_CODE]’ for the method’s source code.

B.4.8 Evaluation Metrics

We evaluated the LLMs’ vulnerability prediction performance using precision, recall, and F1 score metrics. These metrics are particularly suitable for imbalanced datasets in binary classification, offering valuable insights into the models’ effectiveness in identifying vulnerabilities. Additionally, the F1 score is particularly useful in this study because it not only facilitates comparison with previous chapters but also places less emphasis on true negatives, which are of lesser interest in vulnerability prediction tasks, as noted in Subsubsection 4.4.5.3 in Chapter 4.

B.4.9 Approach to Research Question

The experiments in this chapter addressed the research question: *How well do Large Language Models perform on method-level vulnerability prediction tasks?*

To explore this research question, we pursued the following objectives:

1. Evaluate the effectiveness of Large Language Models, particularly ChatGPT, in identifying vulnerabilities within code samples without external context.

2. Compare the performance of Large Language Models with our models in Chapters 4, 5, and 6.

B.4.9.1 Objective 1

Evaluate the effectiveness of Large Language Models, particularly ChatGPT, in identifying vulnerabilities within code samples without external context.

We approached this objective from two perspectives: evaluating the performance of our LLMs using standard evaluation metrics and assessing their performance improvements over time.

We used the OpenAI API's LLMs, specifically GPT-3.5-turbo-0125, GPT-4o-mini, and GPT-4o, to predict vulnerabilities from method-level code samples. The models provided binary classifications indicating the presence or absence of vulnerabilities, which we evaluated using precision, recall, and F1 score metrics.

To compare the LLMs' performance, we completed inter-series and intra-series comparisons. First, we compared the vulnerability prediction performance of the GPT-3 and GPT-4 models, i.e., an inter-series comparison, to assess the improvements between the two series. This analysis highlighted the advancements in the GPT series over time. Next, we evaluated the performance differences within the GPT-4 series models, i.e., intra-series comparison. This comparison showcased the capabilities of the latest models in the GPT-4 series.

For inter-series advancements, we compared the GPT-3.5-turbo-0125 model with the GPT-4o-mini and GPT-4o models, using GPT-3.5-turbo-0125 as a baseline. We performed performance ratio analyses across all evaluation metrics on our within- and mixed-project datasets.

Following the inter-series comparison, we conducted an intra-series analysis to evaluate the vulnerability prediction performance of the GPT-4 series models, using GPT-4o-mini as the baseline. Specifically, we compared GPT-4o-mini and GPT-4o on our within- and mixed-project datasets.

For both inter-series and intra-series comparisons, we calculated performance improvement ratios and percentages to quantify the advancements in the GPT series models.

We calculated the performance improvement ratio PIR as follows:

$$\text{PIR} = \frac{P_e}{P_b}$$

P_e and P_b represent the performance of the model under evaluation and the baseline model for the evaluation metric, respectively.

The performance improvement percentage PIP was then derived as follows:

$$\text{PIP} = (\text{PIR} - 1) \times 100\%$$

These comparisons offered a comprehensive analysis of the vulnerability prediction capabilities of the GPT-3 and GPT-4 series models, highlighting their advancements.

B.4.9.2 Objective 2

Compare the performance of Large Language Models with our models in Chapters 4, 5, and 6.

The second objective was to compare the performance of LLMs with the best-performing models developed in Chapters 4, 5, and 6. We used precision, recall, and F1 score metrics to evaluate the LLMs' vulnerability prediction performance and compared the results with those from the earlier chapters.

This comparison provided insights into the effectiveness of LLMs in vulnerability prediction without external context and highlighted their potential in software security tasks. The following section presents the results from our experiments on both the within- and mixed-project datasets.

B.5 Results

B.5.1 Objective 1 Results

Evaluate the effectiveness of Large Language Models, particularly ChatGPT, in identifying vulnerabilities within code samples without external context.

This objective aimed to assess the effectiveness of LLMs in identifying vulnerabilities in code samples without relying on external context or prior knowledge of known vulnerability patterns.

Table B.1: Large Language Model Performances on the Within-Project Dataset

Metric	GPT-3.5-turbo-0125	GPT-4o-mini	GPT-4o
Precision	0.03006	0.04428	0.04826
Recall	0.12698	0.43915	0.68519
F1 score	0.04861	0.08046	0.09017

Table B.1 shows the performance of GPT-3.5-turbo-0125, GPT-4o-mini, and GPT-4o models on the within-project dataset. The models exhibited varying precision, recall, and F1 scores, reflecting differences in their effectiveness at identifying vulnerabilities. The models demonstrated low performance, particularly in terms of precision and F1 score.

Table B.2: Large Language Model Performances on the Mixed-Project Dataset

Metric	GPT-3.5-turbo-0125	GPT-4o-mini	GPT-4o
Precision	0.12750	0.13406	0.13383
Recall	0.14154	0.36448	0.56614
F1 score	0.13415	0.19602	0.21648

Table B.2 shows the performance of GPT-3.5-turbo-0125, GPT-4o-mini, and GPT-4o models on the mixed-project dataset. The models displayed varying precision, recall, and F1 scores, reflecting differences in their ability to identify vulnerabilities. Performance (based on F1 score) was slightly better on the mixed-project dataset than on the within-project dataset, but precision and F1 scores remained suboptimal.

Metric	**GPT-4o-mini (%)**	**GPT-4o (%)**
Precision	47	61
Recall	246	440
F1 score	66	85

(a) Within-Project Dataset

Metric	**GPT-4o-mini (%)**	**GPT-4o (%)**
Precision	5	5
Recall	158	300
F1 score	46	61

(b) Mixed-Project Dataset

Figure B.4: Performance Improvement Percentages: GPT-4 Over GPT-3 Series (Inter-Series Comparison)

Figure B.4 shows the performance improvement percentages of GPT-4 models compared to the GPT-3 model on the within- and mixed-project datasets, calculated using the *PIP* formula from Subsubsection B.4.9.1. These percentages reflect advancements in the GPT series for out-of-the-box vulnerability prediction tasks.

Recall showed the highest improvement across GPT-4 models on both datasets, while precision had the lowest. The F1 score showed moderate improvement, indicating overall progress between the GPT series.

Metric	**GPT-4o (%)**
Precision	9
Recall	56
F1 score	12

(a) Within-Project Dataset

Metric	**GPT-4o (%)**
Precision	0
Recall	55
F1 score	10

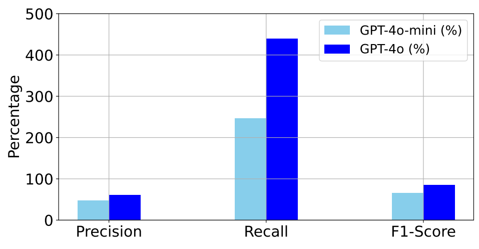
(b) Mixed-Project Dataset

Figure B.5: Performance Improvement Percentages: GPT-4o Over GPT-4o Mini (Intra-Series Comparison)

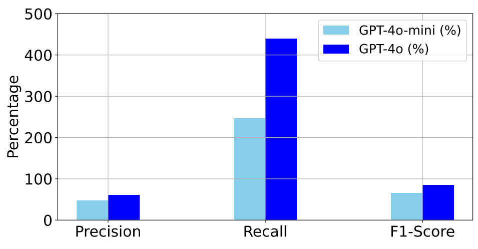
Figure B.5 shows the performance improvement percentages of the GPT-4o model over the GPT-4o-mini model on the within- and mixed-project datasets, calculated using the *PIP* formula from Subsubsection B.4.9.1. These percentages reflect intra-series advancements in the GPT-4 models for vulnerability prediction.

As expected, the improvements were less significant than those in the inter-series comparison, indicating that GPT-4o did not outperform GPT-4o-mini as dramatically as the GPT-4 models outperformed the GPT-3 model, but the improvements were still notable.

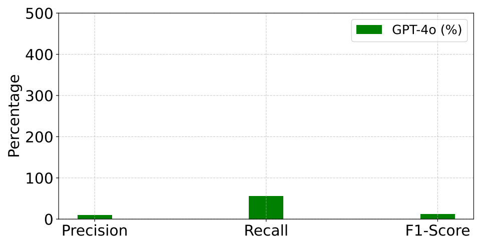
Similar to the inter-series comparison, recall showed the highest improvement between the two GPT-4 models on both datasets, while precision had the lowest. The F1 score showed moderate improvement, indicating a continued upward trend in the GPT series' performance.



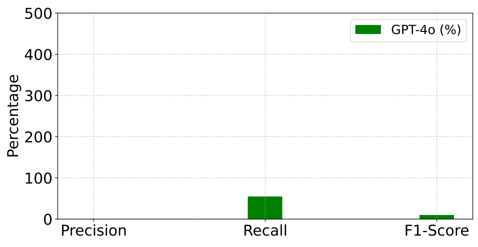
(a) Within-Project Dataset



(b) Mixed-Project Dataset



(c) Within-Project Dataset



(d) Mixed-Project Dataset

Figure B.6: Performance Improvement Percentages: Inter- (GPT-4 Over GPT-3 Series) and Intra-Series (GPT-4o Over GPT-4o Mini) Comparisons

Figure B.6 presents bar charts showing the performance improvement percentages of GPT-4 models over GPT-3 models and GPT-4o over GPT-4o-mini on our within- and mixed-project datasets, based on the data from Tables B.4 and B.5. The bar charts visually highlight the advancements in the GPT series for out-of-the-box vulnerability prediction tasks.

Subfigures B.6a and B.6b (in blue) show the performance improvement percentages of GPT-4 models over the GPT-3 model on the within- and mixed-project datasets, respectively.

Subfigures B.6c and B.6d (in green) show the comparably less pronounced performance improvement percentages of GPT-4o over GPT-4o-mini on the same datasets.

In summary, the LLMs performed suboptimally in identifying vulnerabilities within code samples. All models yielded inadequate results, limiting their practical utility for vulnerability prediction in code samples. However, we observe that the GPT-4 models significantly improve upon GPT-3 models in vulnerability prediction tasks. The inter-series comparison showed substantial improvements, while the intra-series comparison yielded more modest gains, which is expected given that the models belong to the same generation. This upward trend in performance is promising for the future, as it indicates advancements in the GPT series for vulnerability prediction tasks.

B.5.2 Objective 2 Results

Compare the performance of Large Language Models with our models in Chapters 4, 5, and 6.

Table B.3: Within-Project Performance Comparison: Previous Chapters versus Large Language Models

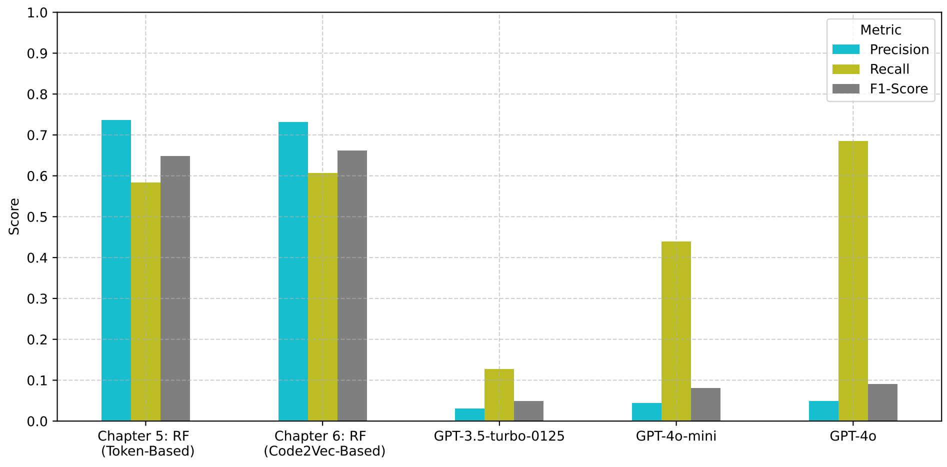
Model	Precision	Recall	F1 score
Chapter 4: RF (Token-Based)	0.73635	0.58345	0.64821
Chapter 5: RF (Code2Vec-Based)	0.73171	0.60668	0.66106
GPT-3.5-turbo-0125	0.03006	0.12698	0.04861
GPT-4o-mini	0.04428	0.43915	0.08046
GPT-4o	0.04826	0.68519	0.09017

Table B.3 presents the within-project performance comparison of our best-performing models from Chapters 4 and 5 with the GPT-3.5-turbo-0125, GPT-4o-mini, and GPT-4o models. The figures for the chapters' models are the pre-hyperparameter tuning values, representing a fair 'out-of-the-box' comparison with the LLMs. Also, we note that 'RF' stands for Random Forest classifier.

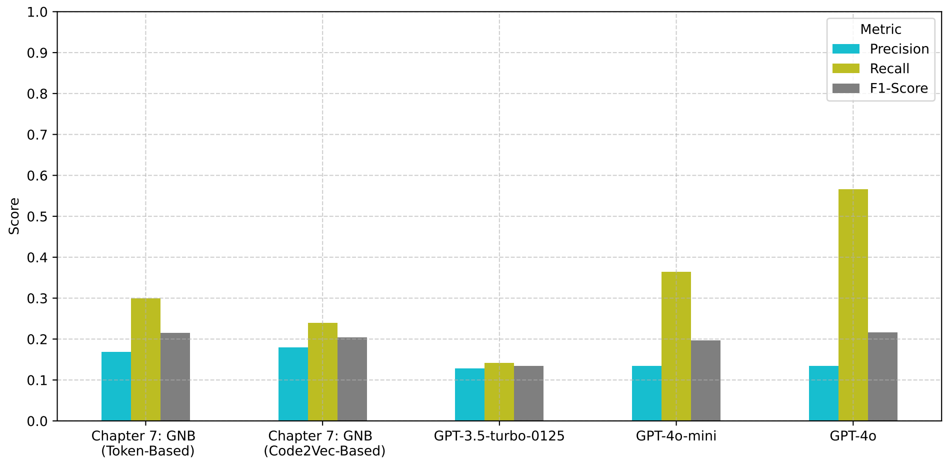
Table B.4 presents a comparison of the mixed-project performance of our best-performing models from Chapter 6 with that of the GPT-3.5-turbo-0125, GPT-4o-mini, and GPT-4o models. Like the within-project comparison, the figures for the chapter's models are pre-hyperparameter tuning. 'GNB' stands for Gaussian Naïve Bayes classifier.

Table B.4: Mixed-Project Performance Comparison: Previous Chapters versus Large Language Models

Model	**Precision**	**Recall**	**F1 score**
Chapter 6: GNB (Token-Based)	0.16780	0.29921	0.21477
Chapter 6: GNB (Code2Vec-Based)	0.17859	0.23869	0.20394
GPT-3.5-turbo-0125	0.12750	0.14154	0.13415
GPT-4o-mini	0.13406	0.36448	0.19602
GPT-4o	0.13383	0.56614	0.21648



(a) Within-Project Performance Comparison



(b) Mixed-Project Performance Comparison

Figure B.7: Performance Comparison: Previous Chapters versus LLM

Figure B.7 shows two grouped bar charts comparing the pre-hyperparameter tuning performance of our best-performing models from Chapters 4, 5, and 6 with the performance of our LLMs.

Subfigure B.7a compares the performance of the models on the within-project dataset, while Subfigure B.7b compares the performance on the mixed-project dataset.

Subfigure B.7a shows the performance of our token-based model (Random Forest Classifier) from Chapter 4, our Code2vec-based model (Random Forest Classifier) from Chapter 5, and the three LLMs on the within-project dataset.

Subfigure B.7b shows the performance of our token-based model (Gaussian Naïve Bayes classifier) from Chapter 6, our Code2vec-based model (Gaussian Naïve Bayes classifier) also from Chapter 6, and the three LLMs on the mixed-project dataset.

The immediate observation from Figure B.7 is that our traditional Random Forest token- and Code2Vec-based models significantly outperform the LLMs in all metrics on the Within-Project dataset.

On the other hand, the LLMs perform competitively with our Gaussian Naïve Bayes token- and Code2Vec-based models on the Mixed-Project dataset, with the GPT-4 models noticeably outperforming the Gaussian Naïve Bayes models in recall. Although all the results in the Mixed-Project dataset are suboptimal.

In summary, our traditional machine learning models outperformed the LLMs in a within-project setting, while the LLMs performed competitively with our traditional models in a mixed-project setting. The LLMs, especially GPT-4o, show the most promise in vulnerability prediction tasks, particularly in recall, but their precision and F1 scores are currently insufficient for practical use.

B.6 Discussion

This study evaluated the effectiveness of LLMs, specifically OpenAI's GPT series, in predicting software vulnerabilities at the method level. The research focused on their out-of-the-box capability to predict vulnerabilities without relying on external context, such as known vulnerability patterns. The results were compared with traditional machine learning models developed in Chapters 4, 5, and 6, and we found that the LLMs' performance was suboptimal in identifying vulnerabilities within code samples without external context. While LLMs show promise in this field, their performance, particularly in terms of precision and F1 score, is insufficient to replace or surpass existing models.

B.6.1 Effectiveness of Large Language Models in Vulnerability Prediction

Using within- and mixed-project datasets, we assessed the GPT-3.5-turbo, GPT-4o-mini, and GPT-4o models. While the GPT-4 series models showed notable improvements in the recall, precision remained low, leading to reduced F1 scores. This indicates that although LLMs can detect more true positives, they also produce many false positives, limiting their practical effectiveness in vulnerability prediction. It also indicates that, based on out-of-the-box performance, LLMs in their current evolution struggle to distinguish subtle differences between vulnerable and non-vulnerable code patterns, resulting in a high rate of false positives.

B.6.2 Comparison of Large Language Models with Previous Models

The comparison of LLMs with traditional machine learning models from Chapters 4, 5, and 6 revealed that the latter significantly outperformed the former in within-project settings. On the other hand, the LLMs performed competitively with the traditional models in mixed-project settings, particularly in recall. We theorise that the traditional models' superior performance in within-project settings is due to their ability to learn from specific vulnerability patterns in the training data, which the LLMs struggle with. In contrast, the LLMs' competitive performance in mixed-project settings is due to their generalisation capabilities, possibly due to their vast pre-trained knowledge.

B.6.3 Performance Improvements in the Generative Pre-trained Transformer Series

The findings highlight clear performance improvements across the GPT series. The shift from GPT-3.5 to GPT-4o-mini and GPT-4o models showed significant gains in the recall, though precision and, consequently, F1 score improvements were modest. This suggests that while LLMs are improving in identifying vulnerabilities, they still need refinement to reduce false positives. The performance ratios between the GPT-3 and GPT-4 series indicate ongoing advancements, particularly in recall, which enhance the models' ability to detect vulnerabilities more effectively.

B.6.4 Limitations of Large Language Models in Vulnerability Prediction

Despite their potential, the LLMs evaluated in this study have notable limitations in vulnerability prediction. The primary issue is their low precision, which results in a high rate of false positives. This could prove costly in software security applications because of the potential for unnecessary alerts and wasted resources in investigating false positives. Additionally, LLMs are highly sensitive to the quality and specificity of prompts, making their effectiveness reliant on well-framed input, a sentiment echoed in most of the reviewed literature in Section B.3.

B.6.5 Context-Aware Large Language Model-based Vulnerability Prediction

Given these limitations, future research could explore the development of context-aware LLMs for vulnerability prediction. These models would use the static knowledge from their training data and dynamically incorporate contextual information, such as the software environment, system architecture, usage patterns, and emerging threats. The fine-tuning feature of OpenAI's LLMs could be leveraged to adapt the models to specific software contexts, enhancing their precision and reducing false positives. This approach could improve precision by filtering irrelevant information and focusing on contextually significant code patterns. Additionally, incorporating real-time feedback could refine predictions and reduce false positives.

Another avenue for creating context-aware LLMs is to leverage Retrieval-Augmented Generation (RAG) models. RAG is an advanced AI framework that integrates traditional information retrieval systems with the capabilities of generative language models. RAG can be integrated into chatbot systems to enhance conversational abilities in practical applications [Rangan and Yin, 2024]. By leveraging external knowledge, RAG-powered chatbots can deliver more comprehensive and context-aware responses, thereby enhancing the user experience. The external knowledge sources enhance the accuracy, relevance, and timeliness of the generated text [Ding et al., 2024].

RAG operates through two main steps. First, it retrieves relevant information from external data sources, such as web pages, knowledge bases, or, in our case, vulnerability databases, using sophisticated search algorithms. The retrieved information is then pre-processed to ensure it is ready for integration [Izacard and Grave, 2020]. In the second step, this information is incorporated into the LLM, enriching the model's understanding and enabling it to generate more precise and contextually relevant responses [Xiong et al., 2024]. RAG employs vector databases to facilitate efficient retrieval based on semantic similarity [Sawarkar et al., 2024].

The primary advantages of RAG include access to up-to-date information, improved factual accuracy, and enhanced contextual relevance. Unlike traditional LLMs, which are limited to pre-trained knowledge, RAG ensures that responses are accurate and current by accessing external sources [Ding et al., 2024]. Because RAG conditions output on retrieved factual information, it promotes consistency and reduces the likelihood of inaccuracies. This is particularly useful in high-precision applications where factual accuracy is paramount, such as vulnerability prediction in software systems. This aspect is crucial in our context, given that our results have demonstrated the need for improved precision in LLM-based vulnerability prediction.

Overall, RAG represents a significant advancement in AI. It combines the strengths of information retrieval and language generation to produce more accurate and contextually grounded outputs, enhancing the performance of LLMs in vulnerability prediction tasks.

B.6.6 Data Challenges in Large Language Model-Based for Vulnerability Prediction

In Subsection 6.2.2, we discussed the significant impact of data quality and quantity on vulnerability prediction models. We categorised these challenges into six themes: data generalisability, data accessibility, data preparation effort, data scarcity, label noise, and data noise. We explained that models often conform to the "garbage in, garbage out" principle, i.e., models are susceptible to the data on which they are trained. Thus, challenges such as the scarcity of high-quality data, inconsistent reporting practices, and organisations' reluctance to share vulnerability data exacerbate these data-related issues.

Although this study shows that advancements in LLMs hold promise for vulnerability prediction, their current poor performance still reflects the same data-related challenges previously discussed. For instance, the need for context-aware LLMs suggests that data preparation will continue to be a significant challenge, requiring substantial expertise and resources. This issue is closely tied to other challenges, such as data generalisability, accessibility, scarcity, and noise. As a result, the trade-offs between using LLMs and traditional machine learning models in vulnerability prediction remain critical in the current generation of LLMs.

B.6.7 Implications

The results of this study provide essential insights into the strengths and limitations of LLMs in predicting vulnerabilities in software code. While LLMs, such as the GPT series, are crucial for text understanding and generation, their application in vulnerability prediction still requires substantial refinement, particularly in terms of precision. Despite their advanced architecture, LLMs may not be entirely suited to vulnerability prediction tasks without significant tuning or additional contextual data. The observed performance improvements across the GPT series indicate progress, but the current state of LLMs warrants caution when using them in critical security tasks.

These findings have significant implications for both researchers and industry practitioners. For researchers, the identified performance gaps suggest that further work is needed to refine LLMs, potentially through hybrid approaches that combine LLMs with traditional models or domain-specific techniques. For industry practitioners, the findings emphasise the need to consider the limitations of LLMs in security-critical applications and the importance of rigorous testing and validation to ensure their reliability and effectiveness.

B.6.8 Recommendations

Based on this study's findings, several recommendations for future research and practice in vulnerability prediction can be made. Firstly, researchers should explore hybrid approaches that combine the strengths of LLMs with traditional machine learning models to improve precision and specificity. Secondly, developing context-aware LLMs that adapt to dynamic software environments and emerging threats is crucial for enhancing real-world effectiveness. Thirdly, practitioners should use LLMs cautiously in security-critical applications, ensuring thorough testing and validation, especially where precision is critical. Additionally, integrating LLMs with other security tools could provide a more balanced approach, leveraging LLMs' strengths in recall while addressing their precision limitations. Additionally, further exploration of context-aware models through fine-tuning or RAGs is recommended to enhance the practical utility of LLMs in software security. Finally, future research should focus on refining the training and tuning processes for LLMs to improve their performance in vulnerability prediction and address the limitations identified in this study.

B.7 Threats to Validity

The findings of this study are subject to several threats to validity that may impact the results and conclusions. These threats are discussed below to provide a comprehensive understanding of the study's limitations and potential biases.

B.7.1 Internal Validity

Internal validity refers to the reliability of this study's results, ensuring they are free from confounding factors.

B.7.1.1 Dataset Characteristics

The data used to evaluate the LLMs poses a significant threat to internal validity. Datasets, especially those in cross- and mixed-project settings, may contain biases or inconsistencies affecting model performance. Noisy labels or incomplete data could lead to inaccurate assessments. Despite efforts to clean the data, eliminating these issues is challenging and may have influenced the results.

B.7.1.2 Dataset Vulnerability versus Non-Vulnerability Distribution

The distribution of vulnerable and non-vulnerable code samples in the datasets can impact model performance. An imbalance in positive and negative samples might skew results, affecting the models' vulnerability prediction capabilities. For example, the within-project dataset has a 1.48% vulnerability rate (Table 4.11), whereas the mixed-project dataset has a 5.28% rate (Table 6.2). As shown in Tables B.1 and B.2, the mixed-project dataset yielded better results, likely due to its higher vulnerability rate, which may have enhanced model performance.

Balancing the datasets could produce different results but also introduce bias, as this adjustment may not accurately reflect real-world scenarios. Moreover, such class distribution adjustments would misalign with the study's aim of evaluating LLMs' out-of-the-box performance.

B.7.1.3 Prompt Design Bias

The effectiveness of LLMs in predicting vulnerabilities can be sensitive to the phrasing and structure of the prompts. Although this study used carefully designed prompts to minimise bias, unintended prompt bias remains possible. Different prompts might lead to varying results, introducing subjectivity that could impact the study's internal validity.

B.7.1.4 Application Programming Interface Request Parameters

The study's request configuration, including temperature and other parameters, could affect LLM performance. Although we followed best practices, these settings introduced a layer of complexity that may have influenced the models' predictions. For instance, the temperature setting, which ranges from 0 (deterministic output) to 1 (more random output), was left at the default value of 0. Changing this setting could alter the LLMs' predictions and introduce bias.

B.7.2 External Validity

External validity concerns the generalisability of the study's findings beyond its specific context.

B.7.2.1 Programming Language Generalisability

One significant threat is the exclusive use of Java datasets. While Java is widely used, the findings may not apply to vulnerability prediction in other programming languages, as distinct syntax and semantic structures can influence the effectiveness of LLMs. Therefore, the results may not generalise to projects in other languages.

B.7.2.2 Granularity of Vulnerability Prediction

The focus on method-level vulnerability prediction may limit the study's generalisability to other levels of granularity. While this granularity enables detailed analysis, the findings may not apply to other levels, such as class-level or file-level vulnerability prediction. The complexity of vulnerabilities varies with the unit of analysis, and LLMs may perform differently in broader contexts.

B.7.2.3 Model Generalisability

The reliance on specific versions of GPT models may limit the generalisability of the results. As LLM technology evolves, newer models might exhibit different strengths and weaknesses. The findings here are based on GPT-3.5-turbo, GPT-

4o-mini, and GPT-4o, and future models may produce different outcomes. Thus, generalising these conclusions to future LLM versions should be done cautiously.

B.7.2.4 Absence of Contextual Information

Our exclusion of external contextual data in the experimental setup may not reflect real-world scenarios where additional context can enhance vulnerability prediction. The absence of this context could limit the applicability of the findings to real-world software security tasks where context-aware analysis is beneficial.

B.8 Conclusion

This study utilised LLMs, specifically OpenAI's GPT series, to predict software vulnerabilities. It aimed to explore the prospect of enhancing software security by leveraging advanced AI models to overcome the limitations of traditional machine learning approaches. The objectives were to assess the out-of-the-box effectiveness of LLMs in vulnerability prediction, compare their performance with that of existing models, and evaluate improvements within the GPT series, focusing on their viability at the method level. Using within-project and mixed-project datasets, the study evaluated LLMs, including GPT-3.5-turbo, GPT-4o-mini, and GPT-4o models, through binary vulnerability prediction classification tasks using precision, recall, and F1 score metrics as evaluation criteria.

Results indicated that while the GPT-4 series improved recall and F1 score, precision remained suboptimal. The LLMs performed better than the traditional models from Chapter 6 on the mixed-project dataset, likely due to their generalisation capabilities facilitated by a vast pre-trained knowledge base, which, according to Yu et al. [2023], comprises billions of parameters for the GPT-3 series and up to a trillion parameters for the GPT-4 series. However, the traditional models from Chapters 4 and 5 significantly outperformed the LLMs on the within-project dataset, suggesting that LLMs struggle with recognising and discerning specific vulnerability patterns within their vast pre-trained knowledge. The findings suggest that despite their advanced architecture, LLMs are not yet optimised for vulnerability prediction and, thus, produce high false positive rates. Thus, significant improvements are needed for LLMs to match or surpass traditional machine learning approaches for vulnerability prediction.

We conclude this study by noting that precision is a significant challenge for LLMs in vulnerability prediction, limiting their practical utility in software security. However, the inter- and intra-generation improvements in the GPT series suggest ongoing advancements that could enhance LLMs' effectiveness in vulnerability prediction, especially on the mixed-project front. Until these advancements are realised, context-aware LLMs attained through fine-tuning or RAG models represent a promising approach to improving LLMs' practical utility in software security.

List of Terms

a priori A Latin term meaning "from the earlier" or "from the cause", which refers to knowledge or information that is deduced from first principles or self-evident propositions.. 152

Abstract Syntax Tree A tree representation of the abstract syntactic structure of source code, which is used for analysis and transformation.. 32

accuracy The proportion of correct predictions made by a classification model.. 37

actionable The usefulness of a vulnerability prediction model in practice, based on an average or above-average level of performance.. 187

anomaly detection The process of identifying patterns in data that do not conform to expected behaviour.. 38

association rule mining A data mining technique that identifies patterns in data, such as frequent itemsets or co-occurrences.. 304

bias A systematic error in a machine learning model that causes it to predict values different from the true values consistently.. 143

binary classification A classification problem where the target variable has two classes, positive and negative.. 33

bug A fault in a computer program that causes it to behave unexpectedly.. 25

bug-fixing change A change made to a software system to fix a bug.. 35

bug-inducing change A change made to a software system that introduces a bug.. 35

churn The number of times a method has been changed in a software system over multiple releases or versions.. 42

class A blueprint for creating objects in many object-oriented programming languages, which defines the properties and behaviours of objects.. 30

class imbalance The situation where one class in a classification problem has significantly more instances than the other class.. 138

classification A type of machine learning problem where the goal is to predict the class of an instance based on its features.. 106

co-evolution The process by which two or more artefacts evolve together, such as bug-fixing and bug-inducing changes.. 35

Code2Vec An AST-based code representation model that learns vector representations of code snippets.. 35

Coefficient of Variation A measure of relative variability used to compare the spread of data sets with different units of measurement.. 257

Common Vulnerabilities and Exposures A list of publicly known cybersecurity vulnerabilities.. 26

Common Weakness Enumeration A list of software weaknesses that lead to vulnerabilities.. 140

correlation analysis A statistical technique that measures the strength and direction of a relationship between two variables.. 151

cross-project prediction The process of predicting software vulnerabilities in a project that is different from the one that the model was trained on.. 237

Cross-Validation A technique used to evaluate the performance of a machine learning model by splitting the dataset into training and testing sets multiple times.. 148

data accessibility The ease with which data can be accessed and used for analysis.. 244

data generalisability See dataset generalisability.. 238

data heterogeneity The extent to which data points in a dataset differ from those in other datasets.. 246

data leakage The situation where information from the test set is inadvertently used to train a machine learning model, leading to overly optimistic performance estimates.. 114

data noise Irrelevant or incorrect data in a dataset that can negatively impact the performance of a machine learning model.. 244

data preparation effort The time and resources required to prepare a dataset for machine learning.. 244

data preprocessing The cleaning and transformation of raw data into a format suitable for machine learning algorithms.. 135

data scarcity The situation where there is an insufficient amount of data to train a machine learning model.. 244

data variability See data heterogeneity.. 238

dataset generalisability The extent to which a dataset can train a model that generalises well to unseen data.. 172

deep learning A subset of machine learning that focuses on artificial neural networks and learning data representations.. 29

defect See bug.. 301

dependent variable A variable being measured in an experiment and affected by the independent variable.. 138

detached bug-inducing artefact A code artefact that contributed to the introduction of a bug but was not involved in its subsequent fix.. 302

diverse Having a variety of different code elements.. 112

domain adaptation The process of transferring knowledge from one domain to another to improve the performance of a machine learning model.. 243

dynamic code analysis The process of analysing source code by executing it to find bugs, security vulnerabilities, or other issues.. 29

error See bug.. 25

F1 score The harmonic mean of precision and recall, which is used to evaluate the performance of a classification model.. 37

fault See bug.. 25

feature An attribute or property of an instance in a dataset that is used to make predictions in a machine learning model.. 29

feature engineering The process of selecting and transforming features in a dataset to improve the performance of a machine learning model.. 44

feature scaling The process of normalising the range of features in a dataset to improve the performance of a machine learning model.. 138

feature selection The process of choosing a subset of features in a dataset to improve the performance of a machine learning model.. 138

flaw See bug.. 25

generalisability The ability of a machine learning model to perform well on unseen data.. 34

generative artificial intelligence A type of artificial intelligence that generates new data, such as images, text, or music.. 335

granularity The unit of analysis in a software system, such as the method-level, class-level, or file-level.. 30

greedy search algorithm An algorithm that makes the best choice at each step to find the optimal solution.. 152

ground truth The actual true values of the target variable in a dataset, which are used to evaluate the performance of a machine learning model.. 137

hit A situation where one or more elements in a target software system method's code representation match those of one or more methods in a vulnerability dataset, facilitated by information retrieval.. 110

hit-dependent metric A metric (feature) type that uses the occurrence of hits between a target software system method's code representation and vulnerability dataset methods to measure and quantify the similarity between them. See hit.. 57

hit-independent metric A metric (feature) type that does not rely on hits but instead relies on the target software system method's code representation alone to measure the intrinsic properties of the method. See hit.. 57

hyperparameter A parameter that is set before the training process of a machine learning model and affects its performance.. 139

hyperparameter tuning The process of selecting the best hyperparameters for a machine learning model to improve its performance.. 139

independent variable A variable manipulated or controlled in an experiment to determine its effect on the dependent variable.. 138

information retrieval The process of obtaining information from a collection of documents, such as web pages or source code.. 30

interpretability The ability to explain and understand how a machine learning model makes predictions.. 168

intricate Having many complexly arranged code elements.. 112

label noise Incorrect or mislabelled instances in a dataset that can negatively impact the performance of a machine learning model.. 244

Large Language Model A language model trained on a large corpus of text data.. 296

machine learning A subset of artificial intelligence that focuses on developing algorithms that allow computers to learn from and make predictions based on data.. 29

method A function or procedure in a programming language, usually defined within a class, that defines the behaviour of an object. See class.. 30

mixed-project A dataset that contains data from multiple software projects.. 32

mixed-project prediction The process of predicting software vulnerabilities using a model trained on a mix of projects.. 238

model convergence The point at which a machine learning model has learned the underlying patterns in the data and stops improving, i.e., where a model's parameters or predictions stabilise during training.. 144

n-gram A contiguous sequence of n items from a given sample of text or speech.. 135

object An instance of a class in object-oriented programming that encapsulates data and methods.. 30

path A sequence of node types in an Abstract Syntax Tree that represents a traversal direction.. 47

path context A core Code2Vec concept that is used to capture the context of an AST path.. 49

pattern matching The process of finding similarities between patterns in data, such as source code.. 30

precision The proportion of true positive predictions out of all positive predictions made by a classification model.. 37

recall The proportion of true positive predictions from all actual positive instances in a dataset.. 37

reinforcement learning A type of machine learning where an agent learns to make decisions by interacting with an environment and receiving rewards or penalties.. 33

Retrieval-Augmented Generation A technique that combines information retrieval and natural language generation to improve the performance of large language models.. 296

security-relevant Having implications for the security of a software system.. 34

Sequential Feature Selection A feature selection technique that selects and combines features one at a time based on the model's performance to find the best subset of features.. 138

shingle A set of n consecutive tokens in a sequence of tokens, often presented as a sliding window or overlapping sequence of all the tokens.. 46

software metric A measure of software characteristics, such as complexity, size, or maintainability.. 34

source code representation A structured way of representing source code, such as ASTs, code embeddings, or n-grams.. 32

static code analysis The process of analysing source code without executing it to find bugs, security vulnerabilities, or other issues.. 29

stress test A test that evaluates the performance of a system under extreme conditions, such as high loads or limited resources.. 238

supervised machine learning A machine learning technique that uses labelled data to train algorithms to predict outcomes. The goal is to create a model to predict the correct output for new data.. 30

Synthetic Minority Oversampling Technique A technique used to balance class distributions in imbalanced datasets.. 138

SZZ Algorithm An algorithm that uses bug-fixing commit information to identify the associated bug-inducing commits.. 301

target software system The software system for which vulnerabilities are being predicted.. 56

token A single programming language element, such as a keyword, operator, or identifier.. 32

transaction database A database that stores transactions, such as purchases or interactions, or in our context, code co-changes among artefacts in a version control system, which are used in association rule mining.. 314

transfer learning A machine learning technique that transfers knowledge from one domain to another to improve the performance of a model.. 243

unsupervised machine learning A type of machine learning where the algorithm is trained on an unlabelled dataset, meaning it is not provided with output labels to learn from.. 33

vulnerability A type of bug with security implications that an attacker can exploit to compromise a software system's security. See bug.. 26

vulnerability dataset A publicly available dataset that contains information and code samples of known software vulnerabilities.. 30

vulnerability detection The process of identifying software vulnerabilities.. 230

vulnerability prediction The process of predicting software vulnerabilities using machine learning models.. 29

vulnerability prediction model A machine learning model that predicts software vulnerabilities.. 29

vulnerability-fixing change See bug-fixing change.. 27

vulnerability-inducing change See bug-inducing change.. 26

vulnerable code pattern A pattern in source code indicative of a software vulnerability.. 39

within-project In the context of software vulnerability prediction, refers to the same project that the model was trained on.. 32

within-project prediction The process of predicting software vulnerabilities within the same project that the model was trained on.. 237

Bibliography

Mohamad Abdolahi and Moreza Zahedh. Sentence matrix normalization using most likely n-grams vector. In *2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, pages 0040–0045. IEEE, 2017.

Vishwanath Akuthota, Raghunandan Kasula, Sabiha Tasnim Sumona, Masud Mohiuddin, Md Tanzim Reza, and Md Mizanur Rahman. Vulnerability Detection and Monitoring Using LLM. In *2023 IEEE 9th International Women in Engineering (WIE) Conference on Electrical and Computer Engineering (WIECON-ECE)*, pages 309–314, nov 2023. doi: 10.1109/WIECON-ECE60392.2023.10456393.

Sadam Al-Azani and El-Sayed M El-Alfy. Using word embedding and ensemble learning for highly imbalanced data sentiment analysis in short arabic text. *Procedia Computer Science*, 109:359–366, 2017.

Fahad Al Debeyan, Tracy Hall, and David Bowes. Improving the performance of code vulnerability prediction using abstract syntax tree information. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 2–11, 2022.

Manar Alohaly and Hassan Takabi. When Do Changes Induce Software Vulnerabilities? In *2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC)*, pages 59–66, oct 2017. doi: 10.1109/CIC.2017.00020.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

Sousuke Amasaki, Tomoyuki Yokogawa, and Aman Hirohisa. An evaluation of word embeddings on vulnerability prediction with software metrics. *WiPiEC Journal-Works in Progress in Embedded Computing Journal*, 9(2), 2023.

Le An and Foutse Khomh. An Empirical Study of Crash-Inducing Commits in Mozilla Firefox. In *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450337151. doi: 10.1145/2810146.2810152. URL `https://doi.org/10.1145/2810146.2810152`.

SS Anju, P Harmya, Noopa Jagadeesh, and R Darsana. Malware detection using assembly code and control flow graph optimization. In *Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India*, pages 1–4. 2010.

Afsah Anwar, Ahmed Abusnaina, Songqing Chen, Frank Li, and David Mohaisen. Cleaning the nvd: Comprehensive quality assessment, improvements, and analyses. *IEEE Transactions on Dependable and Secure Computing*, 19(6):4255–4269, 2021.

B Arivazhagan, S Pandikumar, S Bharani Sethupandian, and R Shankara Subramanian. Pattern discovery and analysis of customer buying behavior using association rules mining algorithm in e-commerce. In *2022 First International Conference on Electrical, Electronics, Information and Communication Technologies (ICEEICT)*, pages 1–5. IEEE, 2022.

Xinbo Ban, Shigang Liu, Chao Chen, and Caslon Chua. A performance evaluation of deep-learnt features for software vulnerability detection. *Concurrency and Computation: Practice and Experience*, 31(19):e5103, 2019.

Ajay Bandi, Pydi Venkata Satya Ramesh Adapa, and Yudu Eswar Vinay Pratap Kumar Kuchi. The power of generative ai: A review of requirements, models, input–output formats, evaluation metrics, and challenges. *Future Internet*, 15(8):260, 2023.

Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

Mathias Berggren, Lisa Kaati, Björn Pelzer, Harald Stiff, Lukas Lundmark, and Nazar Akrami. The generalizability of machine learning models of personality across two text domains. *Personality and Individual Differences*, 217:112465, 2024.

Guru Bhandari, Amara Naseer, and Leon Moonen. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 30–39, 2021.

Zeki Bilgin, Mehmet Akif Ersoy, Elif Ustundag Soykan, Emrah Tomur, Pinar Çomak, and Leyli Karaçay. Vulnerability prediction from source code using machine learning. *IEEE Access*, 8:150672–150684, 2020.

David Binkley, Leon Moonen, and Sibren Isaacman. Featherweight assisted vulnerability discovery. *Information and Software Technology*, 146:106844, 2022.

Michael Franklin Bosu and Stephen G MacDonell. A taxonomy of data quality challenges in empirical software engineering. In *2013 22nd Australian Software Engineering Conference*, pages 97–106. IEEE, 2013.

David Bowes, Tracy Hall, Mark Harman, Yue Jia, Federica Sarro, and Fan Wu. Mutation-aware fault prediction. In *Proceedings of the 25th international symposium on software testing and analysis*, pages 330–341, 2016.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Felivel Camilo, Andrew Meneely, and Meiyappan Nagappan. Do bugs foreshadow vulnerabilities? a study of the chromium project. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 269–279. IEEE, 2015.

Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection. *Information and Software Technology*, 136:106576, 2021.

Yihan Cao, Siyu Li, Yixin Liu, Zhiling Yan, Yutong Dai, Philip S Yu, and Lichao Sun. A comprehensive survey of ai-generated content (aigc): A history of generative ai from gan to chatgpt. *arXiv preprint arXiv:2303.04226*, 2023.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296, 2021.

Chia-Hui Chang and Ching-Chi Hsu. Enabling concept-based relevance feedback for information retrieval on the www. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):595–609, 1999.

Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

Deng Chen, Yan-duo Zhang, Wei Wei, Shi-xun Wang, Ru-bing Huang, Xiao-lin Li, Bin-bin Qu, and Sheng Jiang. Efficient vulnerability detection based on an optimized rule-checking static analysis technique. *Frontiers of Information Technology & Electronic Engineering*, 18(3):332–345, 2017.

Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 519–531, 2022.

Boris Chernis and Rakesh Verma. Machine Learning Methods for Software Vulnerability Detection. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*, IWSPA '18, pages 31–39, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356343. doi: 10.1145/3180445.3180453. URL `http://doi.org/10.1145/3180445.3180453`.

Jitender Kumar Chhabra and Varun Gupta. A survey of dynamic software metrics. *Journal of computer science and technology*, 25(5):1016–1029, 2010.

T Chong, V Anu, and K Z Sultana. Using Software Metrics for Predicting Vulnerable Code-Components: A Study on Java and Python Open Source Projects. In *2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 98–103, aug 2019a. doi: 10.1109/CSE/EUC.2019.00028.

Tai-Yin Chong, Vaibhav Anu, and Kazi Zakia Sultana. Using software metrics for predicting vulnerable code-components: a study on java and python open source projects. In *2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 98–103. IEEE, 2019b.

Pooja Awana Choudhary and Satwinder Singh. Neural network based bug priority prediction model using text classification techniques. *International Journal of Advanced Research in Computer Science*, 8(5), 2017.

Gobinda G Chowdhury. *Introduction to modern information retrieval.* Facet publishing, 2010.

Shaiful Chowdhury, Gias Uddin, Hadi Hemmati, and Reid Holmes. Method-level bug prediction: Problems and promises. *ACM Transactions on Software Engineering and Methodology*, 33(4):1–31, 2024.

Rory Coulter, Qing-Long Han, Lei Pan, Jun Zhang, and Yang Xiang. Code analysis for intelligent cyber systems: A data-driven approach. *Information sciences*, 524: 46–58, 2020.

Roland Croft, Dominic Newlands, Ziyu Chen, and M Ali Babar. An empirical study of rule-based and learning-based approaches for static application security testing. In *Proceedings of the 15th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*, pages 1–12, 2021.

Roland Croft, Yongzheng Xie, and Muhammad Ali Babar. Data preparation for software vulnerability prediction: A systematic literature review. *IEEE Transactions on Software Engineering*, 49(3):1044–1063, 2022.

Mike Czech, Eyke Hüllermeier, Marie-Christine Jakobs, and Heike Wehrheim. Predicting rankings of software verification tools. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics*, pages 23–26, 2017.

Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uira Kulesza, Roberta Coelho, and Ahmed E Hassan. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, jul 2017. ISSN 0098-5589. doi: 10.1109/TSE.2016.2616306.

H K Dam, T Tran, T Pham, S W Ng, J Grundy, and A Ghose. Automatic Feature Learning for Predicting Vulnerable Software Components. *IEEE Transactions on Software Engineering*, 47(1):67–85, jan 2021. ISSN 1939-3520. doi: 10.1109/TSE.2018.2881961.

Debashree Devi, Biswajit Purkayastha, et al. Redundancy-driven modified tomek-link based undersampling: A solution to class imbalance. *Pattern Recognition Letters*, 93:3–12, 2017.

Yujuan Ding, Wenqi Fan, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. A survey on rag meets llms: Towards retrieval-augmented large language models. *arXiv preprint arXiv:2405.06211*, 2024.

Varun Dogra, Sahil Verma, Kavita, Pushpita Chatterjee, Jana Shafi, Jaeyoung Choi, and Muhammad Fazal Ijaz. A complete process of text classification system using state-of-the-art nlp models. *Computational Intelligence and Neuroscience*, 2022(1):1883698, 2022.

Aijuan Dong and Baoying Wang. Domain-based recommendation and retrieval of relevant materials in e-learning. In *2008 IEEE International Workshop on Semantic Computing and Applications*, pages 103–108. IEEE, 2008.

Maureen Doyle and James Walden. An empirical study of the evolution of php web application security. In *2011 Third International Workshop on Security Measurements and Metrics*, pages 11–20. IEEE, 2011.

Gewangzi Du, Liwei Chen, Tongshuai Wu, Chenguang Zhu, and Gang Shi. Cpmsvd: Cross-project multiclass software vulnerability detection via fused deep feature and domain adaptation. In *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4950–4954. IEEE, 2024.

Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 60–71. IEEE, 2019.

Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17:531–577, 2012.

Faris Faisal Fadlalla and Huwaida T Elshoush. Input validation vulnerabilities in web applications: Systematic review, classification, and analysis of the current state-of-the-art. *IEEE Access*, 11:40128–40161, 2023.

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings* *of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324, 2014.

Hantao Feng, Xiaotong Fu, Hongyu Sun, He Wang, and Yuqing Zhang. Efficient vulnerability detection based on abstract syntax tree and deep learning. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 722–727. IEEE, 2020a.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020b.

Javed Ferzund, Syed Nadeem Ahsan, and Franz Wotawa. Software change classification using hunk metrics. In *2009 IEEE International Conference on Software Maintenance*, pages 471–474, 2009. doi: 10.1109/ICSM.2009.5306274.

Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, 33(11):725–743, 2007.

Jaroslav Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. Autofolding for source code summarization. *IEEE Transactions on Software Engineering*, 43(12):1095–1109, 2017.

Georgia Frantzeskou, Stephen MacDonell, Efstathios Stamatatos, and Stefanos Gritzalis. Examining the significance of high-level programming features in source code author classification. *Journal of Systems and Software*, 81(3):447–460, 2008.

Michael Fu and Chakkrit Tantithamthavorn. Linevul: a transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620, 2022.

Sundarakrishnan Ganesh, Tobias Ohlsson, and Francis Palma. Predicting Security Vulnerabilities using Source Code Metrics. In *2021 Swedish Workshop on Data Science (SweDS)*, pages 1–7, dec 2021. doi: 10.1109/SweDS53855.2021.9638 301.

Rodríguez-Pérez Gema, Gregorio Robles, Serebrenik Alexander, Andy Zaidman, Daniel M Germán, and Jesus M Gonzalez-Barahona. How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering*, 25(2):1294–1340, mar 2020. ISSN 13823256. doi: http: //dx.doi.org/10.1007/s10664-019-09781-y.

Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys*, 50(4), 2017. ISSN 15577341. doi: 10.1145/ 3092566. URL `http://doi.org/10.1145/3092566`.

Emanuel Giger, Martin Pinzger, and Harald C Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th working conference on mining software repositories*, pages 83–92, 2011.

Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. Method-level bug prediction. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 171–180. IEEE, 2012.

Shruti Gujral, Gitika Sharma, Sumit Sharma, et al. Classifying bug severity using dictionary based approach. In *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*, pages 599–602. IEEE, 2015.

Junjun Guo, Zhengyuan Wang, Haonan Li, and Yang Xue. Detecting vulnerability in source code using cnn and lstm network. *Soft computing*, 27(2):1131–1141, 2023.

Aakanshi Gupta, Bharti Suri, Vijay Kumar, and Pragyashree Jain. Extracting rules for vulnerabilities detection with static metrics using machine learning. *International Journal of System Assurance Engineering and Management*, 12(1, SI): 65–76, feb 2021. doi: 10.1007/s13198-020-01036-0.

Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 223–226, 2010.

John T Hancock and Taghi M Khoshgoftaar. Survey on categorical data for neural networks. *Journal of Big Data*, 7(1):1–41, 2020.

Hazim Hanif and Sergio Maffeis. Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2022.

Hazim Hanif, Mohd Hairul Nizam Md Nasir, Mohd Faizal Ab Razak, Ahmad Firdaus, and Nor Badrul Anuar. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *Journal of Network and Computer Applications*, 179:103009, 2021.

Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.

Haibo He, Yang Bai, Edwardo A Garcia, and Shutao Li. Adasyn: Adaptive synthetic sampling approach for imbalanced learning. In *2008 IEEE international joint conference on neural networks (IEEE world congress on computational intelligence)*, pages 1322–1328. IEEE, 2008.

Enrique Herrera-Viedma, Javier López Gijón, Sergio Alonso, Josefina Vilchez, Concha Garcia, Luis Villén, and Antonio Gabriel López-Herrera. Applying aggregation operators for information access systems: An application in digital libraries. *International Journal of Intelligent Systems*, 23(12):1235–1250, 2008.

Toby D Hocking, Joseph R Barr, and Tyler Thatcher. Interpretable linear models for predicting security vulnerabilities in source code. In *2022 Fourth International Conference on Transdisciplinary AI (TransAI)*, pages 149–155. IEEE, 2022.

Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th international conference on Software engineering*, pages 392–411, 1992.

Aram Hovsepyan, Riccardo Scandariato, Wouter Joosen, and James Walden. *Software Vulnerability Prediction using Text Analysis Techniques.* 2012. ISBN 9781450315081. URL `http://www.fortify.com/`.

Aram Hovsepyan, Riccardo Scandariato, and Wouter Joosen. Is newer always better? the case of vulnerability prediction models. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–6, 2016.

Vinay M Igure and Ronald D Williams. Taxonomies of attacks and vulnerabilities in computer systems. *IEEE Communications Surveys & Tutorials*, 10(1):6–19, 2008.

Gautier Izacard and Edouard Grave. Leveraging passage retrieval with generative models for open domain question answering. *arXiv preprint arXiv:2007.01282*, 2020.

Hui Jiang, Linfeng Song, Yubin Ge, Fandong Meng, Junfeng Yao, and Jinsong Su. An ast structure enhanced decoder for code generation. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 30:468–476, 2021.

Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 135–146. IEEE, 2017.

Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. Vulnerability prediction models: A case study on the linux kernel. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–10. IEEE, 2016.

Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. The importance of accounting for real-world labelling when predicting software vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 695–705, 2019.

Rong Jin, Hamed Valizadegan, and Hang Li. Ranking refinement and its application to information retrieval. In *Proceedings of the 17th international conference on World Wide Web*, pages 397–406, 2008.

Zhaoyan Jin and Yang Yu. Current and future research of machine learning based vulnerability detection. In *2018 Eighth International Conference on Instrumentation & Measurement, Computer, Communication and Control (IMCCC)*, pages 1562–1566. IEEE, 2018.

Ilias Kalouptsoglou, Miltiadis Siavvas, Dimitrios Tsoukalas, and Dionysios Kehagias. Cross-project vulnerability prediction based on software metrics and deep learning. In *Computational Science and Its Applications–ICCSA 2020: 20th International Conference, Cagliari, Italy, July 1–4, 2020, Proceedings, Part IV 20*, pages 877–893. Springer, 2020.

Ilias Kalouptsoglou, Miltiadis Siavvas, Dionysios Kehagias, Alexandros Chatzigeorgiou, and Apostolos Ampatzoglou. Examining the capacity of text mining and software metrics in vulnerability prediction. *Entropy*, 24(5):651, 2022.

Ilias Kalouptsoglou, Miltiadis Siavvas, Apostolos Ampatzoglou, Dionysios Kehagias, and Alexander Chatzigeorgiou. Software vulnerability prediction: A systematic mapping study. *Information and Software Technology*, page 107303, 2023.

Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE transactions on software engineering*, 28(7):654–670, 2002.

Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4401–4410, 2019.

Arvinder Kaur and Ruchikaa Nayyar. A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. *Procedia Computer Science*, 171:2023–2029, 2020.

Faiza Khan, Summrina Kanwal, Sultan Alamri, and Bushra Mumtaz. Hyper-parameter optimization of classifiers, using an artificial immune network and its application to software bug prediction. *IEEE Access*, 8:20954–20964, 2020.

Sunghun Kim, Thomas Zimmermann, Kai Pan, and E James Jr. Whitehead. Automatic Identification of Bug-Introducing Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 81–90, 2006. doi: 10.1109/ASE.2006.23.

Sunghun Kim, E James Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on software engineering*, 34(2):181–196, 2008.

K Knill and S Young. Hidden markov models in speech and language processing. In *Corpus-based methods in language and speech processing*, pages 27–68. Springer, 1997.

Mei Kobayashi and Koichi Takeda. Information retrieval on the web. *ACM computing surveys (CSUR)*, 32(2):144–173, 2000.

Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. Pathminer: a library for mining of path-based representations of code. In

2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pages 13–17. IEEE, 2019.

Triet Huynh Minh Le and M Ali Babar. On the use of fine-grained vulnerable code statements for software vulnerability assessment models. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 621–633, 2022.

Niklas Leicht, Ivo Blohm, and Jan Marco Leimeister. Leveraging the power of the crowd for software testing. *IEEE Software*, 34(2):62–69, 2017.

Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE international conference on software maintenance and evolution (ICSME)*, pages 249–260. IEEE, 2017.

Runhao Li, Chao Feng, Xing Zhang, and Chaojing Tang. A lightweight assisted vulnerability discovery method using deep neural networks. *IEEE Access*, 7: 80079–80092, 2019.

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*, pages 292–303, 2021.

Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis. 2016. doi: 10.1145/2991079.2991102. URL `http://dx.doi.org/10.1145/2991079.2991102`.

G Lin, S Wen, Q L. Han, J Zhang, and Y Xiang. Software Vulnerability Detection Using Deep Neural Networks: A Survey. *Proceedings of the IEEE*, 108(10): 1825–1848, oct 2020a. ISSN 1558-2256. doi: 10.1109/JPROC.2020.2993293.

Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Xiang Yang, Olivier De Vel, and Paul Montague. Cross-Project Transfer Representation Learning for Vulnerable Function Discovery. *IEEE Transactions on Industrial Informatics*, 14(7):3289–3297, 2018. ISSN 15513203. doi: http://dx.doi.org/10.1109/TII.2018.2821768.

Guanjun Lin, Sheng Wen, Qing Long Han, Jun Zhang, and Yang Xiang. Software Vulnerability Detection Using Deep Neural Networks: A Survey. *Proceedings of the IEEE*, 108(10):1825–1848, 2020b. ISSN 15582256. doi: 10.1109/JPROC.20 20.2993293.

Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. An Empirical Study on Android-Related Vulnerabilities. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 2–13. IEEE Press, 2017. ISBN 9781538615447. doi: 10.1109/MSR.2017.60. URL `https://doi.org/10.1109/MSR.2017.60`.

Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. An empirical study on the effectiveness of static c code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pages 544–555, 2022.

Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. A unified multi-task learning model for ast-level and token-level code completion. *Empirical Software Engineering*, 27(4):91, 2022a.

Guang-Hong Liu, Gang Wu, Zheng Tao, Jian-Mei Shuai, and Zhuo-Chun Tang. Vulnerability analysis for x86 executables using genetic algorithm and fuzzing. In *2008 Third International Conference on Convergence and Hybrid Information Technology*, volume 2, pages 491–497. IEEE, 2008.

Shigang Liu, Guanjun Lin, Qing-Long Han, Sheng Wen, Jun Zhang, and Yang Xiang. Deepbalance: Deep-learning and fuzzy oversampling for vulnerability detection. *IEEE Transactions on Fuzzy Systems*, 28(7):1329–1343, 2019.

Shigang Liu, Guanjun Lin, Lizhen Qu, Jun Zhang, Olivier De Vel, Paul Montague, and Yang Xiang. Cd-vuld: Cross-domain vulnerability discovery based on deep domain adaptation. *IEEE Transactions on Dependable and Secure Computing*, 19(1):438–451, 2022b. doi: 10.1109/TDSC.2020.2984505.

Francesco Lomio, Emanuele Iannone, Andrea De Lucia, Fabio Palomba, and Valentina Lenarduzzi. Just-in-time software vulnerability detection: Are we there yet? *Journal of Systems and Software*, page 111283, 2022.

Kasper Luckow, Rody Kersten, and Corina Pasareanu. Complexity vulnerability analysis using symbolic execution. *Software Testing, Verification and Reliability*, 30(7-8):e1716, 2020.

Yu Luo, Weifeng Xu, and Dianxiang Xu. Compact abstract graphs for detecting code vulnerability with gnn models. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 497–507, 2022.

Yuanhua Lv and ChengXiang Zhai. Adaptive relevance feedback in information retrieval. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 255–264, 2009.

Ruchika Malhotra and Shweta Meena. Empirical validation of machine learning techniques for heterogeneous cross-project change prediction and within-project change prediction. *Journal of Computational Science*, 76:102230, 2024.

Y Malik, C R S Campos, and F Jaafar. Detecting Android Security Vulnerabilities Using Machine Learning and System Calls Analysis. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 109–113, jul 2019. doi: 10.1109/QRS-C.2019.00033.

Christopher D Manning. *Introduction to information retrieval.* Syngress Publishing,, 2008.

Melvin E Maron. On indexing, retrieval and the meaning of about. *Journal of the american society for information science*, 28(1):38–43, 1977.

Melvin Earl Maron and John Larry Kuhns. On relevance, probabilistic indexing and information retrieval. *Journal of the ACM (JACM)*, 7(3):216–244, 1960.

Alejandro Mazuera-Rozo, Anamaria Mojica-Hanke, Mario Linares-Vásquez, and Gabriele Bavota. Shallow or deep? an empirical study on detecting vulnerabilities using deep learning. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 276–287. IEEE, 2021.

Nadia Medeiros, Naghmeh Ivaki, Pedro Costa, and Marco Vieira. An Empirical Study On Software Metrics and Machine Learning to Identify Untrustworthy Code. In *2021 17th European Dependable Computing Conference (EDCC)*, pages 87–94, 2021. doi: 10.1109/EDCC53658.2021.00020.

Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodríguez Tejeda, Matthew Mokary, and Brian Spates. When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 65–74, oct 2013. doi: 10.1109/ESEM.2013.19.

Robert Chisolm Miller. *A type-checking preprocessor for Cilk 2, a multithreaded C language*. PhD thesis, Massachusetts Institute of Technology, 1995.

Roweida Mohammed, Jumanah Rawashdeh, and Malak Abdullah. Machine learning with oversampling and undersampling techniques: overview study and experimental results. In *2020 11th international conference on information and communication systems (ICICS)*, pages 243–248. IEEE, 2020.

Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. Challenges with applying vulnerability prediction models. *ACM International Conference Proceeding Series*, 21-22-Apri, 2015. doi: 10.1145/2746194.2746198.

Sara Moshtari, Ashkan Sami, and Mahdi Azimi. Using complexity metrics to improve software security. *Computer Fraud & Security*, 2013(5):8–17, 2013.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

Rebecca Moussa, Danielle Azar, and Federica Sarro. Investigating the use of one-class support vector machine for software defect prediction. *arXiv preprint arXiv:2202.12074*, 2022.

Nuthan Munaiah, Felivel Camilo, Wesley Wigham, Andrew Meneely, and Meiyappan Nagappan. Do bugs foreshadow vulnerabilities? an in-depth study of the chromium project. *Empirical Software Engineering*, 22:1305–1347, 2017.

Md Nadim et al. *Investigating the Techniques to Detect and Reduce Bug Inducing Commits During Change Operations in Software Systems*. PhD thesis, University of Saskatchewan, 2020.

Tarak Nandy, Mohd Yamani Idna Bin Idris, Rafidah Md Noor, Laiha Mat Kiah, Lau Sian Lun, Nor Badrul Annuar Juma'at, Ismail Ahmedy, Norjihan Abdul Ghani, and Sananda Bhattacharyya. Review on security of internet of things authentication mechanism. *IEEE Access*, 7:151054–151089, 2019.

Kollin Napier, Tanmay Bhowmik, and Shaowei Wang. An empirical study of text-based machine learning models for vulnerability detection. *Empirical Software Engineering*, 28(2):38, 2023.

Mohammad Taneem Bin Nazim, Md Jobair Hossain Faruk, Hossain Shahriar, Md Abdullah Khan, Mohammad Masum, Nazmus Sakib, and Fan Wu. Systematic analysis of deep learning model for vulnerable code detection. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1768–1773. IEEE, 2022.

Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540, 2007.

Van Nguyen, Trung Le, Olivier de Vel, Paul Montague, John Grundy, and Dinh Phung. Dual-component deep domain adaptation: A new approach for cross project software vulnerability detection. In *Advances in Knowledge Discovery and Data Mining: 24th Pacific-Asia Conference, PAKDD 2020, Singapore, May 11–14, 2020, Proceedings, Part I 24*, pages 699–711. Springer, 2020.

Van Nguyen, Trung Le, Chakkrit Tantithamthavorn, John Grundy, and Dinh Phung. Deep Domain Adaptation With Max-Margin Principle for Cross-Project Imbalanced Software Vulnerability Detection. *ACM Trans. Softw. Eng. Methodol.*, 33(6), jun 2024. ISSN 1049-331X. doi: 10.1145/3664602. URL `https://doi.org/10.1145/3664602`.

Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. Regvd: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 178–182, 2022.

Sho Ogino, Yoshiki Higo, and Shinji Kusumoto. Evaluating Bug Prediction under Realistic Settings. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 491–495, mar 2021. doi: 10.1109/SANER50967.2021.00052.

OWASP. Owasp top ten web application security risks| owasp. 2021. URL `https://owasp.org/www-project-top-ten/`.

Liang Pang, Yanyan Lan, Jiafeng Guo, Jun Xu, Jingfang Xu, and Xueqi Cheng. Deeprank: A new deep architecture for relevance ranking in information retrieval. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 257–266, 2017.

Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimilano Di Penta, Denys Poshynanyk, and Andrea De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *2013*

35th International conference on software engineering (ICSE), pages 522–531. IEEE, 2013.

Garrett Partenza, Trevor Amburgey, Lin Deng, Josh Dehlinger, and Suranjan Chakraborty. Automatic identification of vulnerable code: Investigations with an ast-based neural network. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1475–1482. IEEE, 2021.

Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Re-evaluating method-level bug prediction. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 592–601, mar 2018. doi: 10.1109/SANER.2018.8330264.

Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

José D'Abruzzo Pereira, João R Campos, and Marco Vieira. Machine Learning to Combine Static Analysis Alerts with Software Metrics to Detect Security Vulnerabilities: An Empirical Study. In *2021 17th European Dependable Computing Conference (EDCC)*, pages 1–8, 2021. doi: 10.1109/EDCC53658.2021.00008.

Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 426–437, 2015.

J Pokropiński, J Gasiorek, P Kramarczyk, and L Madeyski. SZZ Unleashed-RA-C: An Improved Implementation of the SZZ Algorithm and Empirical Comparison with Existing Open Source Solutions. *Studies in Systems, Decision and Control*, 377:181–199, 2022. ISSN 21984182. doi: 10.1007/978-3-030-77916-0_7.

Tommy Pollock. Reducing human error in cyber security using the human factors analysis classification system (hfacs). 2017.

Rishi Rabheru, Hazim Hanif, and Sergio Maffeis. A hybrid graph neural network approach for detecting php vulnerabilities. In *2022 IEEE Conference on Dependable and Secure Computing (DSC)*, pages 1–9. IEEE, 2022.

Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 432–441. IEEE, 2013.

Keshav Rangan and Yiqiao Yin. A fine-tuning enhanced rag system with quantized influence measure as ai judge. *arXiv preprint arXiv:2402.17081*, 2024.

Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 428–439, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884848. URL `https://doi.org/10.1145/2884781.2884848`.

Sofia Reis and Rui Abreu. A ground-truth dataset of real security patches. *arXiv preprint arXiv:2110.09635*, 2021.

Douglas A Reynolds et al. Gaussian mixture models. *Encyclopedia of biometrics*, 741(659-663), 2009.

Timothé Riom, Arthur Sawadogo, Kevin Allix, Tegawendé F Bissyandé, Naouel Moha, and Jacques Klein. Revisiting the vccfinder approach for the identification of vulnerability-contributing commits. *Empirical Software Engineering*, 26:1–30, 2021.

Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE, 2018.

Sefa Eren Şahin, Ecem Mine Özyedierler, and Ayse Tosun. Predicting vulnerability inducing function versions using node embeddings and graph neural networks. *Information and Software Technology*, 145:106822, 2022.

Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th international conference on software engineering*, pages 1157–1168, 2016.

Gerard Salton. Introduction to modern information retrieval. *McGraw-Hill*, 1983.

Hazem Peter Samoaa, Firas Bayram, Pasquale Salza, and Philipp Leitner. A systematic mapping study of source code representation for deep learning in software engineering. *IET Software*, 16(4):351–385, 2022.

Tefko Saracevic. Evaluation of evaluation in information retrieval. In *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 138–146, 1995.

Indah Mayang Sari, Dedy Rahman Wijaya, Wahyu Hidayat, and Rathimala Kannan. An approach to classify rice quality using electronic nose dataset-based naïve bayes classifier. In *2021 International Symposium on Electronics and Smart Devices (ISESD)*, pages 1–5. IEEE, 2021.

Sobhan Sarkar, Nikhil Khatedi, Anima Pramanik, and J Maiti. An ensemble learning-based undersampling technique for handling class-imbalance problem. In *Proceedings of ICETIT 2019: Emerging Trends in Information Technology*, pages 586–595. Springer, 2020.

Kunal Sawarkar, Abhilasha Mangal, and Shivam Raj Solanki. Blended rag: Improving rag (retriever-augmented generation) accuracy with semantic search and hybrid query-based retrievers. *arXiv preprint arXiv:2404.07220*, 2024.

Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014.

Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge, 2008.

Hossain Shahriar and Mohammad Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Computing Surveys (CSUR)*, 44(3): 1–46, 2012.

Nowrin Muhaimin Shailee, Api Alam, Tanvir Ahmed, Rifat Al Mamun Rudro, and Kamruddin Nur. Software bug prediction using machine learning on jm1 dataset. In *2024 International Conference on Advances in Computing, Communication, Electrical, and Smart Systems (iCACCESS)*, pages 01–06. IEEE, 2024.

Zhidong Shen and Si Chen. A survey of automatic software vulnerability detection, program repair, and defect prediction techniques. *Security and Communication Networks*, 2020(1):8858010, 2020.

Yonghee Shin and Laurie Williams. Is complexity really the enemy of software security? In *Proceedings of the 4th ACM workshop on Quality of protection*, pages 47–50, 2008a.

Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 315–317, 2008b.

Yonghee Shin and Laurie Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18(1):25–59, feb 2013. ISSN 13823256. doi: http://dx.doi.org/10.1007/s10664-011-9190-8.

Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on software engineering*, 37(6):772–787, 2010.

Shivkumar Shivaji, E James Whitehead Jr., Ram Akella, and Sunghun Kim. Reducing Features to Improve Bug Prediction. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 600–604, USA, 2009. IEEE Computer Society. ISBN 9780769538914. doi: 10.1109/ASE.2009.76. URL `https://doi.org/10.1109/ASE.2009.76`.

Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2012.

Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.

Rui Shu, Tianpei Xia, Laurie Williams, and Tim Menzies. Dazzle: using optimized generative adversarial networks to address security data class imbalance issue. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 144–155, 2022.

Miltiadis Siavvas, Erol Gelenbe, Dionysios Kehagias, and Dimitrios Tzovaras. Static analysis-based approaches for secure software development. In *Security in Computer and Information Sciences: First International ISCIS Security Workshop 2018, Euro-CYBERSEC 2018, London, UK, February 26-27, 2018, Revised Selected Papers 1*, pages 142–157. Springer International Publishing, 2018.

Fatimah Sidi, Payam Hassany Shariat Panahy, Lilly Suriani Affendey, Marzanah A Jabar, Hamidah Ibrahim, and Aida Mustapha. Data quality: A survey of data quality dimensions. In *2012 International Conference on Information Retrieval & Knowledge Management*, pages 300–304. IEEE, 2012.

Gurdev Singh, Dilbag Singh, and Vikram Singh. A study of software metrics. *IJCEM International Journal of Computational Engineering & Management*, 11 (2011):22–27, 2011.

Sandeep Singh. Analysis of bug tracking tools. *International Journal of Scientific & Engineering Research*, 4(7):134–140, 2013.

Shashank Kumar Singh and Amrita Chaturvedi. Applying deep learning for discovery and analysis of software vulnerabilities: A brief survey. *Soft Computing: Theories and Applications: Proceedings of SoCTA 2019*, pages 649–658, 2020.

Amit Singhal et al. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

Peter Sommerlad, Guido Zgraggen, Thomas Corbat, and Lukas Felber. Retaining comments when refactoring code. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 653–662, 2008.

Fei Song and W Bruce Croft. A general language model for information retrieval. In *Proceedings of the eighth international conference on Information and knowledge management*, pages 316–321, 1999.

Quach Sophia, Lamothe Maxime, Kamei Yasutaka, and Shang Weiyi. An empirical study on the use of SZZ for identifying inducing changes of non-functional bugs. *Empirical Software Engineering*, 26(4), jul 2021. ISSN 13823256. doi: http://dx.doi.org/10.1007/s10664-021-09970-8.

Georgios Spanos and Lefteris Angelis. A multi-target approach to estimate software vulnerability characteristics and severity scores. *Journal of Systems and Software*, 146:152–166, 2018.

Jeffrey Stuckman, James Walden, and Riccardo Scandariato. The effect of dimensionality reduction on software vulnerability prediction models. *IEEE Transactions on Reliability*, 66(1):17–37, 2016.

Kazi Zakia Sultana. Towards a software vulnerability prediction model using traceable code patterns and software metrics, 2017a.

Kazi Zakia Sultana. Towards a software vulnerability prediction model using traceable code patterns and software metrics. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1022–1025. IEEE, 2017b.

Kazi Zakia Sultana and Tai-Yin Chong. A proposed approach to build an automated software security assessment framework using mined patterns and metrics. In *2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 176–181. IEEE, 2019.

Kazi Zakia Sultana and Byron J Williams. Evaluating micro patterns and software metrics in vulnerability prediction, 2017.

Kazi Zakia Sultana, Byron J Williams, and Amiangshu Bosu. A Comparison of Nano-Patterns vs. Software Metrics in Vulnerability Prediction, 2018a.

Kazi Zakia Sultana, Byron J Williams, and Amiangshu Bosu. A comparison of nano-patterns vs. software metrics in vulnerability prediction. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 355–364. IEEE, 2018b.

Kazi Zakia Sultana, Vaibhav Anu, and Tai-Yin Chong. Using software metrics for predicting vulnerable classes and methods in Java projects: A machine learning approach. *Journal of Software (Online)*, 33(3), mar 2021. doi: http://dx.doi.org /10.1002/smr.2303.

Kazi Zakia Sultana, Charles B Boyd, and Byron J Williams. A software vulnerability prediction model using traceable code patterns and software metrics. *SN Computer Science*, 4(5):599, 2023.

Zoltán Szabó and Vilmos Bilicki. A New Approach to Web Application Security: Utilizing GPT Language Models for Source Code Inspection. *Future Internet*, 15(10):326, 2023. URL `https://www.proquest.com/scholarly-journals/new-approach-web-application-security-utilizing/docview/2882511389/se-2?accountid=14511`.

Heping Tang, Shuguang Huang, Yongliang Li, and Lei Bao. Dynamic taint analysis for vulnerability exploits detection. In *2010 2nd International Conference on Computer Engineering and Technology*, volume 2, pages V2–215. IEEE, 2010.

Wei Tang, Mingwei Tang, Minchao Ban, Ziguo Zhao, and Mingjun Feng. Csgvd: A deep learning approach combining sequence and graph embedding for source code vulnerability detection. *Journal of Systems and Software*, 199:111623, 2023.

Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, Akinori Ihara, and Kenichi Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 812–823. IEEE, 2015.

Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2016.

Christopher Theisen and Laurie Williams. Better together: Comparing vulnerability prediction models. *Information and Software Technology*, 119:106204, 2020.

Philipp Thölke, Yorguin-Jose Mantilla-Ramos, Hamza Abdelhedi, Charlotte Maschke, Arthur Dehgan, Yann Harel, Anirudha Kemtur, Loubna Mekki Berrada, Myriam Sahraoui, Tammy Young, et al. Class imbalance should not throw you off balance: Choosing the right classifiers and performance metrics for brain decoding with imbalanced data. *NeuroImage*, 277:120253, 2023.

Riom Timothé, Arthur Sawadogo, Kevin Allix, Tegawendé F Bissyandé, Moha Naouel, and Jacques Klein. Revisiting the VCCFinder approach for the identification of vulnerability-contributing commits. *Empirical Software Engineering*, 26(3), may 2021. ISSN 13823256. doi: http://dx.doi.org/10.1007/s10664-021-09944-w.

Christos Tjortjis. Mining association rules from code (marc) to support legacy software management. *Software Quality Journal*, 28(2):633–662, 2020.

Ivan Tomek. Two modifications of cnn. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-6(11):769–772, 1976. doi: 10.1109/TSMC.1976.4309452.

Arun Veeramani, Kausik Venkatesan, and K Nalinadevi. Abstract syntax tree based unified modeling language to object oriented code conversion. In *Proceedings of the 2014 International Conference on Interdisciplinary Advances in Applied Computing*, pages 1–8, 2014.

Andreas Vogelsang and Jannik Fischbach. Using large language models for natural language processing tasks in requirements engineering: A systematic guideline. *arXiv preprint arXiv:2402.13823*, 2024.

James Walden, Jeff Stuckman, and Riccardo Scandariato. Predicting vulnerable components: Software metrics vs text mining. In *2014 IEEE 25th international symposium on software reliability engineering*, pages 23–33. IEEE, 2014.

H Wang, G Ye, Z Tang, S H Tan, S Huang, D Fang, Y Feng, L Bian, and Z Wang. Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection. *IEEE Transactions on Information Forensics and Security*, 16:1943–1958, 2021a. ISSN 1556-6021. doi: 10.1109/TIFS.2020.3044773.

Zhengyuan Wang, Junjun Guo, and Haonan Li. Vulnerability Feature Extraction Model for Source Code Based on Deep Learning. In *2021 International Conference on Computer Network, Electronic and Automation (ICCNEA)*, pages 21–25, 2021b. doi: 10.1109/ICCNEA53019.2021.00016.

L Wartschinski, Y Noller, T Vogel, T Kehrer, and L Grunske. VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python.

Information and Software Technology, 144, 2022. ISSN 09505849. doi: 10.1016/j.infsof.2021.106809.

Huihui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI*, pages 3034–3040, 2017.

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*, pages 364–374. IEEE, 2009.

Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big data*, 3:1–40, 2016.

Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. Exploring and Exploiting the Correlations between Bug-Inducing and Bug-Fixing Commits. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 326–337. ACM, 2019. ISBN 9781450355728. doi: 10.1145/3338906.3338962.

Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. Exploring parameter-efficient fine-tuning techniques for code generation with large language models. *arXiv preprint arXiv:2308.10462*, 2023.

Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 87–98, 2016.

Garrett Wilson and Diane J Cook. A survey of unsupervised deep domain adaptation. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 11(5): 1–46, 2020.

Bolun Wu, Futai Zou, Ping Yi, Yue Wu, and Liang Zhang. Slicedlocator: Code vulnerability locator based on sliced dependence graph. *Computers & Security*, 134:103469, 2023.

Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*, 2023.

Lei Xiao, Hao Zhong, Jianjian Liu, Kaiyu Zhang, Qizhen Xu, and Le Chang. A novel source code representation approach based on multi-head attention. *Electronics*, 13(11):2111, 2024.

Haoyi Xiong, Jiang Bian, Yuchen Li, Xuhong Li, Mengnan Du, Shuaiqiang Wang, Dawei Yin, and Sumi Helal. When search engine services meet large language models: Visions and challenges. *arXiv preprint arXiv:2407.00128*, 2024.

Akihisa Yamada and Osamu Mizuno. A Text Filtering Based Approach to Classify Bug Injected and Fixed Changes. In *2014 IIAI 3rd International Conference on Advanced Applied Informatics*, pages 680–686, aug 2014. doi: 10.1109/IIAI-AAI.2014.141.

Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499–510, 2013.

Chenyang Yang, Rachel A Brower-Sinning, Grace Lewis, and Christian Kästner. Data leakage in notebooks: Static detection and better processes. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

Limin Yang, Xiangxue Li, and Yu Yu. VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pages 1–7, dec 2017. doi: 10.1109/GLOCOM.2017.8254428.

Xinli Yang, Jingjing Liu, and Denghui Zhang. A comprehensive taxonomy for prediction models in software engineering. *Information*, 14(2):111, 2023.

Recep Yıldırım, Kerem Aydın, and Orçun Çetin. Evaluating the impact of conventional code analysis against large language models in api vulnerability detection. In *European Interdisciplinary Cybersecurity Conference*, pages 57–64, 2024.

A A Younis and Y K Malaiya. Using Software Structure to Predict Vulnerability Exploitation Potential. In *2014 IEEE Eighth International Conference on Software Security and Reliability-Companion*, pages 13–18, jun 2014. doi: 10.1109/SERE-C.2014.17.

Awad Younis, Yashwant Malaiya, Charles Anderson, and Indrajit Ray. To fear or not to fear that is the question: Code characteristics of a vulnerable functionwith an existing exploit. In *Proceedings of the sixth ACM conference on data and application security and privacy*, pages 97–104, 2016.

Ping Yu, Hua Xu, Xia Hu, and Chao Deng. Leveraging generative ai and large language models: a comprehensive roadmap for healthcare integration. In *Healthcare*, volume 11, page 2776. MDPI, 2023.

Chaoning Zhang, Chenshuang Zhang, Sheng Zheng, Yu Qiao, Chenghao Li, Mengchun Zhang, Sumit Kumar Dam, Chu Myaet Thwal, Ye Lin Tun, Le Luang Huy, et al. A complete survey on generative ai (aigc): Is chatgpt from gpt-4 to gpt-5 all you need? *arXiv preprint arXiv:2303.11717*, 2023a.

Chunyong Zhang, Bin Liu, Yang Xin, and Liangwei Yao. CPVD: Cross Project Vulnerability Detection Based on Graph Attention Network and Domain Adaptation. *IEEE Transactions on Software Engineering*, 49(8):4152–4168, aug 2023b. ISSN 1939-3520. doi: 10.1109/TSE.2023.3285910.

Dongping Zhang, Hequn Xian, Jiyang Chen, and Zhiguo Xu. VDCNet: A Vulnerability Detection and Classification System in Cross-Project Scenarios. In L Iliadis, A Papaleonidas, P Angelov, and C Jayne, editors, *ARTIFICIAL NEURAL*

NETWORKS AND MACHINE LEARNING, ICANN 2023, PT I, volume 14254 of *Lecture Notes in Computer Science*, pages 305–316, 2023c. ISBN 978-3-031-44206-3; 978-3-031-44207-0. doi: 10.1007/978-3-031-44207-0_26.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

Tianhang Zhang, Qingfeng Du, Jincheng Xu, Jiechu Li, and Xiaojun Li. Software defect prediction and localization with attention-based models and ensemble learning. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pages 81–90. IEEE, 2020.

Ting Zhang, Ivana Clairine Irsan, Ferdian Thung, and David Lo. Cupid: Leveraging chatgpt for more accurate duplicate bug report detection. *arXiv preprint arXiv:2308.10022*, 2023d.

Ting Zhang, Ivana Clairine Irsan, Ferdian Thung, and David Lo. Revisiting sentiment analysis for software engineering in the era of large language models. *arXiv preprint arXiv:2310.11113*, 2023e.

Xin Zhang, Hongyu Sun, Zhipeng He, MianXue Gu, Jingyu Feng, and Yuqing Zhang. Vdbwgdl: Vulnerability detection based on weight graph and deep learning. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 186–190. IEEE, 2022.

Yan-Ping Zhang, Li-Na Zhang, and Yong-Cheng Wang. Cluster-based majority under-sampling approaches for class imbalance learning. In *2010 2nd IEEE International Conference on Information and Financial Engineering*, pages 400–404. IEEE, 2010.

Zhehao Zhao, Bo Yang, Ge Li, Huai Liu, and Zhi Jin. Precise learning of source code contextual semantics via hierarchical dependence structure and graph attention networks. *Journal of Systems and Software*, 184:111108, 2022.

Alice Zheng and Amanda Casari. *Feature engineering for machine learning: principles and techniques for data scientists.* " O'Reilly Media, Inc.", 2018.

Wei Zheng, Jialiang Gao, Xiaoxue Wu, Fengyu Liu, Yuxing Xun, Guoliang Liu, and Xiang Chen. The impact factors on the performance of machine learning-based vulnerability detection: A comparative study. *Journal of Systems and Software*, 168:110659, 2020.

Andy Zhou, Kazi Zakia Sultana, and Bharath K Samanthula. Investigating the Changes in Software Metrics after Vulnerability is Fixed. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 5658–5663, dec 2021. doi: 10.1109/BigData52589.2021.9671334.

Daihong Zhou, Yijian Wu, Lu Xiao, Yuanfang Cai, Xin Peng, Jinrong Fan, Lu Huang, and Heng Chen. Understanding evolutionary coupling by fine-grained co-change relationship analysis. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 271–282. IEEE, 2019a.

Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Thanh Le-Cong, Junda He, Bach Le, and David Lo. Patchzero: Zero-shot automatic patch correctness assessment. *arXiv preprint arXiv:2303.00202*, 2023.

Xin Zhou, Ting Zhang, and David Lo. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24, pages 47–51, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705007. doi: 10.1145/3639476.3639762. URL `https://doi.org/10.1145/3639476.3639762`.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019b.

Ziyi Zhou, Huiqun Yu, and Guisheng Fan. Effective approaches to combining lexical and syntactical information for code summarization. *Software: Practice and Experience*, 50(12):2313–2336, 2020.

Yufan Zhuang, Sahil Suneja, Veronika Thost, Giacomo Domeniconi, Alessandro Morari, and Jim Laredo. Software vulnerability detection via deep learning over disaggregated code graph representation. *arXiv preprint arXiv:2109.03341*, 2021.

Noah Ziems and Shaoen Wu. Security Vulnerability Detection Using Deep Learning Natural Language Processing, may 2021.

Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 Third international conference on software testing, verification and validation*, pages 421–428. IEEE, 2010.

Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. $\mu\mu$VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2224–2236, 2021. ISSN 1941-0018. doi: 10.1109/TDSC.2019.2942930.