Beyond CVEs

An Analysis of Untracked Software Vulnerabilities Disclosed in Public Issue Trackers

EPA Master Thesis

Mădălin Simion

Delft University of Technology

TU Delft

Beyond CVEs

An Analysis of Untracked Software Vulnerabilities Disclosed in Public Issue Trackers

by

Mădălin Simion

Student Name	Student Number
Mădălin Simion	5838363

First Supervisor:	Yury Zhauniarovich
Second Supervisor:	Sepinoud Azimi Rashti
Chair:	Michel van Eeten
Submission date:	July 2024
Faculty:	Technology, Policy and Management, Delft

Cover:	Image created by Microsoft Copilot Designer
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Executive Summary

In today's digital world, software vulnerabilities pose serious security risks, often leading to severe data breaches and system compromises as an outcome of their exploitation. This thesis, titled "An Analysis of Software Vulnerabilities Disclosed in Public Issue Trackers on GitHub" addresses a critical gap in our understanding of untracked software vulnerabilities in open-source projects on GitHub—vulnerabilities that are discovered and published publicly without formal recognition and tracking through standardized vulnerability tracking systems. One important tracking system investigated in this research is the CVE (Common Vulnerabilities and Exposures) system.

Why should we care about these untracked vulnerabilities? Open-source software is the backbone of many technological innovations and services we rely on daily. Untracked vulnerabilities are published in public issue trackers of these projects, trackers to which both good and bad actors have access and the vulnerabilities are usually unpatched. In this way, they can serve as a clear recipe both for the developers, who can release a fix for the issue in a cause, but also for potential hackers, who can use the issue description, sometimes accompanied by a proof of concept exploitation, to disrupt the normal use of the applications. This research quantifies the prevalence and severity of these untracked security vulnerabilities, providing a clearer picture of the risks involved by answering the question: *What is the extent of vulnerabilities in open-source projects on GitHub that do not have CVE identification numbers?*

The first step of the research was to train a model for security issue classification. For this task, several transformer models have been employed. The DeBERTaV3 model, finetuned on the Chromium dataset, demonstrated superior performance in classifying security issues, achieving an F1 score of 0.9 after threshold adjustment. This represents a 15% improvement in performance over well-known models like FARSEC, which recorded an F1 score of 0.78. The second step of the research was to apply the fine-tuned model on a different project, gRPC, to investigate the cross-project applicability of the model. Through the application of the DeBERTaV3 model on the project case study, it was found that 2.4% of the total issues in the issue tracker were predicted to be security-related. Out of these, 52 issues were validated as potential CVEs by a security expert, compared to only 11 listed in the NVD (National Vulnerability Database). This stark contrast reveals significant underreporting of vulnerabilities and gives an indication of the scale of the aforementioned untracked vulnerabilities.

What are the implications of this underreporting? Without a robust system to identify and report these vulnerabilities, open-source software remains at high risk of exploitation. This thesis advocates for improving current vulnerability disclosure protocols in open-source communities. By establishing a more coordinated and secure disclosure process, where sensitive information is shared privately between developers and project maintainers, the risk of public exposure and exploitation can be minimized. The root cause of these untracked vulnerabilities being present in issue trackers is the developers who post such sensitive information publicly, rather than choosing private communication channels. From the literature, it is not so clear why this behavior occurs, however understanding their motivations is crucial for designing effective policies that encourage responsible disclosure. A first policy advice is provided in this thesis, focusing on improving coordination between two crucial actors of the disclosure process, vulnerability discoverers and project maintainers. Future research should focus on the motivations of developers to foster a culture of secure and responsible vulnerability management.

While the CVE system was not designed to be exhaustive, this research makes a significant contribution by highlighting that there are more vulnerabilities in a project than those that are tracked and proposing practical enhancements to strengthen open-source software security. The insights and recommendations presented support the need for better communication between the actors involved in the coordinated vulnerability disclosure process. Finally, the empirical results aim to inform decision-makers in developing targeted and effective policies that will contribute to the creation of safer and more reliable digital products for everyone.

Contents

Glossary

Abbreviation	Definition
AI	Artificial Intelligence
ALBERT	A Lite BERT for Self-supervised Learning of Language Representations
API	Application Programming Interface
Bug	A bug is defined in this research as an error or flaw in a computer program that causes it to produce incorrect or unexpected results or to behave in unintended ways.
BERT	Bidirectional Encoder Representations from Transformers
BLSTM	Bidirectional Long Short-Term Memory
CNA	Coordinating Numbering Authority
CNN	Convolutional Neural Network
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
LSTM	Long Short-Term Memory
ML	Machine Learning
NLP	Natural Language Processing
NSBR	Non-Security Bug Report
ROC	Receiver Operating Characteristic
RoBERTa	Robustly Optimized BERT Pretraining Approach
Security issue	A security issue is a bug in software that can have security implications, such as requiring a feature request for enhanced security or addressing existing security protocols
SVM	Support Vector Machine
Untracked vulnerabilities	Vulnerabilities that have been identified and published in issue trackers but have not been tracked in common databases using tracking IDs such as CVE IDs
Vulnerability	A software vulnerability is a specific flaw or weakness in a software system that can be exploited by a cyber attacker to gain unauthorized access or perform unauthorized actions

1 Introduction

1.1. Background

An important aspect of modern digital technology is represented by open-source software, which in the past decades has become an integral part of the global technology landscape [1], with adoption growing due to the cost-saving and innovation speed benefits. It is estimated that throughout commercial products, 85% to 97% of the software used comes from open-source projects [2]. As the reliance on these projects grows, so does the importance of ensuring their security, especially due to the inherent availability and deployment, which makes it attractive for bad actors and others who are interested in exploiting vulnerabilities [3]. Companies but also local governments continue to run software that is exposed to vulnerabilities [4]. Furthermore, organizations may be running vulnerable software because they do not acquire information about exposed software from vulnerability databases such as NVD or VulnDB [5].

Open Source Software (OSS) is developed on collaborative platforms such as GitLab [6], BitBucket [7], SourceForge [8] and GitHub [9], with the latter being the most used platform, hosting popular projects such as Tensorflow [10] and Docker [11]. These platforms offer an open environment for collaboration in software development and version control. The open nature of OSS code ensures universal accessibility, fostering a community-centric approach that significantly enhances the speed of feedback and iteration. This environment encourages an active exchange of ideas, where bugs and feature requests are swiftly reported, deliberated, and patches are often more rapidly developed compared to closed-source projects, where feedback mechanisms may be constrained to a select group of internal stakeholders [12]. However, the transparency that accelerates innovation and collaboration can also reveal security vulnerabilities to the public eye. Between the discussions of new features and bug fixes, there might be issues that tie to security vulnerabilities. These disclosures are accessible to everyone, including malicious actors who may exploit this information to craft targeted attacks against the software. While this openness facilitates rapid remediation by the community, it also requires vigilance to prevent exploitation by those with nefarious intentions.

A feature of these development platforms is that each project has a section known as an issue tracker, or bug tracker, that serves as a collaborative space for tracking problems, enhancements, tasks, and other discussions related to the project's development. In this section, users can create, comment, and prioritize issues, making it a centralized hub for communication among contributors. To prioritize the issues and have a better overview of incoming requests, the developers classify these issues manually by assigning different issue labels.

Some projects may choose to create a private communication channel between developers (issue finder) and product maintainers (vendor) to communicate vulnerability-related information. This path for information communication would be considered part of the coordinated vulnerability disclosure process (CVD) [13]. A coordinated vulnerability disclosure involves a private reporting channel where security issues can be submitted by contributors or external security researchers to the project maintainers. By implementing such a protocol, project maintainers can control the flow of sensitive information

to prevent premature exposure of vulnerabilities to potential attackers [13]. Once a vulnerability is reported, the project team assesses the risk, develops a fix, and then strategically releases an update to the community, often accompanied by a detailed security advisory and a tracking number. This approach ensures that vulnerabilities are addressed and patched before the details are made public, minimizing the window of opportunity for exploitation. The discoverer of a security vulnerability or the software vendor can further decide to take a security issue forward and coordinate with a Coordinating Numbering Authority (CNA) [14], either GitHub or external, to obtain a CVE number. Although the CVD process is recommended as the best practice for managing security vulnerabilities, developers sometimes still disclose sensitive information in public issue trackers. This practice is problematic because it exposes vulnerabilities before patches are available, potentially leading to zero-day attacks. Since these issues are made public outside the CVD process and do not receive a CVE number, they often attract less attention than those vulnerabilities that are tracked and cataloged with CVE identifiers.

It is important to note that there are multiple sources for structured data on tracked vulnerabilities and exposures, including the National Vulnerability Database, China National Vulnerability Database (CNDB), Zero Science Lab and others [15]. For this thesis, when referring to tracked vulnerabilities, only the NVD has been chosen as a data source rather than investigating multiple available databases.

1.2. Research Problem

The research problem addressed in this work is the lack of attention given to untracked vulnerabilities in open-source software repositories. Despite the significant role that issue trackers on platforms like GitHub play in the collaborative development process, there exists a substantial gap in the systematic tracking and coordination of vulnerabilities. Many vulnerabilities remain unassigned with Common Vulnerabilities and Exposures numbers, leaving them untracked and obscure within the vast sea of bug reports. This oversight results in an unclear understanding of the extent to which such vulnerabilities are present and active within project issue trackers. Consequently, these overlooked vulnerabilities can easily become prime targets for bad actors, who may exploit them to compromise the security of the software, leading to potential breaches that pose serious risks to users and organizations.

1.3. Research Gaps

The research gaps identified below are based on the review of the relevant literature in the field of vulnerability management described in Chapter 2.

There is a knowledge gap concerning the performance of transformer models for security issue classification. Previous studies have extensively examined automatic issue classification using various methods. Nonetheless, there appears to be a lack of investigation into the application of transformer models for classifying security issues. Given the advanced capabilities of transformer models in deep learning, they are anticipated to significantly enhance performance over traditional methods when applied to datasets previously examined by other researchers such as the Chromium dataset.

There is a knowledge gap in regards to the extent of untracked vulnerabilities published in public issue trackers. The area of untracked vulnerabilities disclosed in issue trackers remains unexplored. While the detection of vulnerabilities from codebases is well-documented, the specific exploration of issue trackers has largely concentrated on identifying security issues, rather than vulnerabilities. Moreover, the models developed for security issue classification have rarely been applied to projects beyond those used for initial training, which limits their practical utility. This oversight potentially leaves a substantial number of untracked vulnerabilities undetected in issue trackers, increasing the risk of exploitation without corresponding CVE numbers.

1.4. Research Objective

This research seeks to illuminate the scale of this issue and provide insights into improving vulnerability management practices within open-source ecosystems. By investigating open-source software security issues on platforms like GitHub, this research seeks to fill a knowledge gap related to the development of secure software products. At the heart of this investigation is the recognition that open-source plays an important role in the current digital economy [16]. The security of OSS is paramount, especially due to the dependency of the software industry on open-source software, which can represent

a weak link targeted by supply chain attacks [17]. Through this research, more clarity will be brought to how many untracked security vulnerabilities are present in a large project on GitHub by creating a pipeline to semi-automatically extract, predict and validate vulnerabilities from the issue tracker.

1.5. Research Question

The research questions are built on the research gap and research objective, and in this thesis we aim to answer:

> *What is the extent of vulnerabilities in open-source projects on GitHub that do not have CVE identification numbers?*

The research primarily aims to enhance the domain of automatic security issue classification and, concurrently, delve deeper than existing studies by investigating the hidden vulnerabilities and their potential impacts. The subquestions that further delineate the scope of the main research question are targeted to specific areas:

1. **Effectiveness of Transformer Models**: Can transformer models be effectively applied to security issue classification, and how does their performance compare to established classification models documented in the literature?
2. **Prevalence of Untracked Vulnerabilities**: How prevalent are untracked vulnerabilities in widely-used C++ projects on GitHub, and what are their common characteristics?
3. **Impact of Hidden Vulnerabilities**: What is the potential impact of these hidden vulnerabilities on the security of open-source software?

1.6. Research Methods and Results

To answer the research question, a pipeline to semi-automatically extract, predict, and validate vulnerabilities from GitHub's issue tracker has been established. For predicting vulnerabilities, transformer models based on BERT [18], RoBERTa [19], and DeBERTaV3 [20] have been fine-tuned on two versions of the Chromium dataset presented by Wu et al. [21] and have unveiled notable findings. DeBERTaV3 emerged as the most effective model, achieving an f1 score of 0.45 with impressive recal of 0.98 and precision rates of 0.30, respectively. Moreover, adjusting the classification threshold to 0.95 altered the precision-recall tradeoff, enhancing the f1 score to 0.83, driven by a recall of 0.94 and a precision of 0.75. Additionally, the exploration of Gemma [22] and Mistral [23] with few-shot-prompting revealed their limitations due to insufficient information from the base models, as reflected in their low f1 scores of 0.11 and 0.12. These results underscore the critical role of dataset quality in the performance of deep learning models applied to cybersecurity, suggesting that larger, more robust datasets can significantly improve outcomes.

The fine-tuned DeBERTav3 model has subsequently been used to predict security issues from the gRPC project on GitHub. From a total of 9906 issues, the model predicted 232 to be security-related. Upon further validation with a security expert, 52 issues have been proven to be hidden vulnerabilities. For gRPC, only 11 CVEs are published on the NVD database at the time of writing [24]. This means that only 17% of the total vulnerabilities (tracked and untracked) received CVE numbers. These results emphasize that there are more vulnerabilities out in the wild and available to the public than the ones that currently receive CVEs. With only 17% of identified vulnerabilities being assigned CVE numbers, there is a significant risk that many issues are not sufficiently addressed or patched, leaving systems potentially open to exploitation. Furthermore, organizations using software versions vulnerable to untracked vulnerabilities may not be notified about these risks, leaving them susceptible to security threats. Without such notifications, they are at risk of not upgrading to newer, patched versions of the software. This gap can provide attackers with more opportunities to exploit undisclosed or underreported vulnerabilities. The insights derived from this research are positioned to influence future efforts in the cybersecurity domain, potentially leading to more effective and harmonized vulnerability management practices across digital platforms.

While the results of this research are promising and illustrate the prevalence of untracked vulnerabilities within issue trackers, several limitations must be considered when interpreting the findings, as detailed in Section 5.2. Notably, the model's fine-tuning and application were confined to the C++ programming

language. This specificity may introduce a bias by presuming that issues are homogenous across different programming languages, potentially skewing the results when applied to other languages. Hence, applying the model to different projects might necessitate further fine-tuning using language-specific labeled datasets. Additionally, the relatively small size of the dataset could restrict the model's ability to identify security issues not encountered during the fine-tuning phase, but that might appear in other projects' issue trackers. Nevertheless, such issues are acknowledged in the field of vulnerability identification and have been presented in previous research [25].

1.7. Cybersecurity as a Grand Challenge

Cybersecurity has emerged as a critical issue on a global scale, deeply intertwined with economic stability, national security, and individual safety. The presence of digital technology across all sectors of modern society has exponentially increased the complexity of cybersecurity as a field. Not only does it involve multiple actors—from individual hackers to state-sponsored groups—it also crosses national borders, making any single approach to securing digital infrastructure inadequate [26]. The threats posed by cybersecurity breaches are not merely technical but also have significant economic and national security implications. Cybersecurity is no longer a concern limited to IT departments but is a strategic issue that affects all levels of an organization and indeed entire nations. This complex web of concerns means that addressing cybersecurity effectively requires coordinated actions across various domains—from legal frameworks and corporate governance to technical defenses and international cooperation [26, 27]. Cybersecurity's importance has escalated as our reliance on digital systems has grown. These systems are not only ubiquitous but also form the backbone of critical infrastructure in multiple sectors, including finance, healthcare, and utilities. The rapid evolution of cyber threats, which are becoming increasingly sophisticated, poses a challenge for existing defense mechanisms, which often lag behind the capabilities of attackers [28]. This ongoing battle against cyber threats is compounded by the dynamic nature of the digital landscape. Cybersecurity is an evolving problem where new vulnerabilities and methods of attack are constantly developed. As such, it is a field marked by continuous adaptation and requires a proactive approach to defense and mitigation strategies [27]. To be succinct, cybersecurity has grown over time into a grand challenge. A problem involving multiple actors, spanning across the globe and without a clear solution.

The process of coordinating vulnerability fixes is part of the grand challenge of cybersecurity due to its inherently complex and multi-faceted nature. It involves numerous stakeholders [14], each with their own roles, responsibilities, and perspectives (see Section 2.6). The coordination process must navigate varying interests, capabilities, and priorities among these parties, such as differing policies of CNAs, the response times of software vendors, and the technical and security needs of end-users. Moreover, the ever-evolving landscape of cybersecurity threats adds another layer of complexity, making it imperative to adapt quickly and efficiently to new vulnerabilities while ensuring accurate and timely communication among all involved parties. This dynamic and intricate environment poses significant challenges in achieving effective and harmonized vulnerability management. However, it is presumed that the vulnerability coordination is not perfect, and some developers do not go through the suggested disclosure path of involving the CNA and communicating with the maintainers privately, but disclose security issues in the public issue tracker, exposing potential critical vulnerabilities to the public.

The issue of vulnerabilities in open-source projects extends beyond the open-source community itself. It has been observed that security researchers and developers of security tools primarily focus on vulnerabilities assigned with CVE numbers [14, 3, 29], often neglecting those without them. This oversight is compounded by the fact that many developers may not initiate the process of obtaining CVE numbers for every discovered vulnerability, by not connecting to the CNA, or the software vendor (see Figure 2.2). This can lead to vulnerabilities being stuck in the issue tracker. Consequently, this leads to a significant gap in the notification and awareness of users, who may continue to operate systems that contain untracked vulnerabilities.

The significance of addressing security issues in open-source software cannot be overstated. Open-source projects are susceptible to vulnerability exploits, even more than closed-source projects because potential attackers can view the code, reducing the effort needed for exploit development [29]. Exploitation of vulnerabilities within these projects can have profound implications for both the companies relying on such software and the broader public, as explained in the following section.

Companies integrating open-source software into their products and services face potential threats if security vulnerabilities go unaddressed. A successful exploitation of a vulnerability can lead to data breaches, service disruptions, financial losses, and damage to the company's reputation [30]. The interconnected nature of modern software ecosystems means that a vulnerability in one component can have cascading effects on entire systems [17]. Even if the vulnerability is corrected, the downstream software products are still at risk of being exploited until they incorporate the correction and release a patch [29].

A significant example is the Apache Struts vulnerability CVE-2017-5638, which was a Remote Code Execution (RCE) flaw in the Apache Struts 2 framework, a popular open-source framework for developing web applications in Java [31]. The vulnerability was due to an improper handling of a certain kind of error message, which allowed attackers to execute arbitrary Java code on a server running an application built with Struts 2. The vulnerability was first reported to the Apache Struts team on March 7, 2017, and a security advisory along with a patch was released promptly [31]. However, prior to this official report, there were discussions and code changes related to the area of the vulnerability [31]. The vulnerability was widely exploited in the wild following its public disclosure, leading to significant negative impacts, such as the Equifax data breach, where attackers exploited this vulnerability to compromise customer data of up to 143 million Equifax customers [31].

Beyond the corporate realm, the broader public is also at risk when security issues in open-source software are left unattended. Public services, critical infrastructure, and applications used by individuals may rely on open-source components. Exploitation of vulnerabilities in these components can compromise the privacy, safety, and well-being of users. For example, attacks on healthcare systems or critical infrastructure could have severe consequences for public health and safety.

1.7.1. EPA Revelance

The relevance of my thesis to the Engineering and Policy Analysis (EPA) master program is significant and aligns closely with the core requirements and objectives of the program. The proposed addresses critical aspects of decision-making quality in the context of a grand societal challenge—cybersecurity. This challenge is embedded in the socio-economic and political environments of our digital age, making the research highly pertinent to the EPA program.

The thesis embodies an analytical character, by employing deep learning techniques to analyze security issues, and providing an examination of the security issues to determine the prevalence and impact of untracked CVE vulnerabilities. This analytical approach is evidenced by the development and fine-tuning of transformer models like DeBERTaV3, which achieved significant improvements in performance metrics for security issue classification. The propoesed research methodology in Chapter 3 systematically looks into the problem of hidden vulnerabilities, applies rigorous analytical tools, and interprets the results to offer meaningful insights into the cybersecurity landscape.

The research adopts a systems perspective by examining the interplay between different actors within the open-source software ecosystem. It highlights the roles of developers, project maintainers, and security experts, emphasizing the need for better coordination and communication among these stakeholders. The multi-actor perspective aids in understanding the complex dynamics of vulnerability disclosure and management. This thesis takes a first step in improving the coordination of vulnerabilites between stakeholders by proposing improvements to the disclosure of vulnerabilities between project developers and maintainers.

The thesis makes systematic use of EPA methods and techniques for problem analysis and exploration. It integrates conceptual modeling and simulation techniques to predict and validate security vulnerabilities in GitHub repositories. The research employs a semi-automatic pipeline to extract, predict, and validate vulnerabilities, utilizing transformer models trained on datasets from Chromium and gRPC projects. This methodological rigor ensures that the research findings are robust, generalizable, and applicable to real-world scenarios.

Cybersecurity is identified as a grand societal challenge, with implications for public and private sectors alike. The thesis provides empirical evidence of the underreporting of vulnerabilities and the associated risks, thereby informing decision-makers about the need for enhanced vulnerability management practices. The research contributes to the development of targeted policies and interventions aimed

at improving the security of open-source software, which is a critical component of modern digital infrastructure. By addressing the gap in the CVE system and proposing practical solutions, the thesis directly supports decision-makers in creating safer and more reliable digital products.

The relevance of the research to public policy is evident in its potential to influence policy decisions at the interface between public and private domains. The insights gained from the analysis of untracked vulnerabilities in open-source software can guide policymakers in developing frameworks that promote secure software practices and responsible vulnerability disclosure. This, in turn, can help mitigate the risks associated with cybersecurity threats, thereby enhancing the overall resilience of digital systems that underpin critical public services and infrastructure.

In summary, my thesis aligns closely with the EPA program by demonstrating a strong analytical approach, incorporating systems and multi-actor perspectives, utilizing EPA-specific methods and techniques, addressing a grand societal challenge, and providing actionable insights for decision-makers in the public policy domain.

1.8. Structure of the Thesis

This thesis is organized into chapters, sections, and subsections to facilitate a logical progression of ideas and ease of navigation. Within these subdivisions, the document incorporates various types of content including plain text, illustrative figures, informative graphs, and detailed tables. To succinctly address the research questions, separate content boxes are placed throughout the text, providing concise summaries of the answers. Furthermore, paragraphs discussing the implications of the findings are distinguished by vertical sidebars. These sidebars serve a crucial role, offering qualitative insights into the implications of the research findings and elucidating their broader significance within the larger context of the study.

This thesis is structured to unfold the problem of untracked security vulnerabilities in open-source software. It begins with Chapter 2, Literature Review, which critically examines existing scholarly work concerning the detection, reporting, and management of software vulnerabilities. This chapter highlights gaps in current methodologies, particularly the oversight of vulnerabilities that do not receive CVE numbers, and sets the academic groundwork for the subsequent application of the methodology. Chapter 3, Methodology, introduces the specific datasets utilized—primarily sourced from GitHub—along with the criteria for their selection. It details the approach of applying advanced machine learning techniques, specifically transformer models to classify security issues. This chapter not only describes the operational framework of the study but also underscores the aspect of employing natural language processing (NLP) tools for security classification. In Chapter 4, Results, the outcomes of the empirical analysis are presented. This chapter provides a comparative analysis of the performance metrics of various transformer models, showcasing their efficacy and limitations in identifying and classifying security threats within OSS. Detailed statistics and model evaluations discuss how different settings and thresholds influence the accuracy and reliability of vulnerability detection, offering insights into the practical applications of these findings. Furthermore, the details on the validation of the vulnerabilities by experts are presented. The culmination of the thesis is presented in Chapter 5, Conclusion, which synthesizes the insights gained throughout the research. It reflects on the scholarly and practical contributions of the study, proposing recommendations for future research and for enhancing the CVE system. The conclusion also calls for a more integrated approach to vulnerability management that includes but extends beyond the traditional CVE tracking systems. The chapter ends with acknowledgment of research limitations and proposal of future work.

2

Literature Review

2.1. Software Vulnerabilities

Software vulnerabilities refer to the weaknesses or flaws within a software system that create potential security risks, exposing the system to unauthorized access or harmful activities [14]. These vulnerabilities can be the result of coding errors, design flaws, or inadequate security measures within the software [32]. The presence of such vulnerabilities provides an avenue for attackers to exploit the system, leading to potential data breaches, system failures, or other malicious activities that result in damages costing companies millions of dollars annually [33]. The exploitation of software vulnerabilities can have significant consequences, ranging from data theft and system compromise to widespread disruption of services [34]. Therefore, the identification and remediation of software vulnerabilities are essential steps in enhancing the security posture of software systems and safeguarding against potential cyber threats [35, 1].

As can be seen in the classification of software bugs depicted in figure 2.1, bugs can be security issues or non-security related bugs. Security issues can be vulnerabilities or non-vulnerabilities, such as security feature requests. Software vulnerabilities are classified into two categories: detected and undetected [3]. Furthermore, detected vulnerabilities, can be published, on issue trackers or relevant websites, or unpublished. More interestingly, published vulnerabilities can be coordinated amongst stakeholders, using CVE numbers, or remain untracked and living in the issue tracker of a project.

Issue tracker systems are widely used by software developers to manage bug reports. For large projects, the trackers contain tens if not hundreds of thousands of reports, and fixing them is time-consuming. In the issue trackers, around 3% of the issues are estimated to be security related [36]. Manually identifying security bugs from reports in bug-tracking systems is difficult and error-prone [37], due to the relative complexity of the issues compared to normal bugs. The security issues are also more difficult to resolve, as Bühlmann & Ghafari [38] observed that discussions related to security issues progress slower than that of non-security issues and that developers are slower in concluding a security issue compared to a non-security bug [38]. Researchers have been focusing on automating the process of security issues identification, in order to improve the security of the software products. A recent mapping study of publications using bug reports concluded that at the time of the study, 42 papers were written on vulnerability classification [39]. For example, Das et al. [40] proposed a machine-learning-based approach to identify security bugs. Other researchers proposed to use semi-supervised learning models [41]. Deep learning was also showing promising results for the classification of vulnerabilities, however, Chakraborty et al. [25] in their research demonstrated that in a real-world vulnerability prediction scenario, some of the proposed models had a decrease in performance of up to 50%, failing to classify issues as security related, or giving false positives. As the field of vulnerability management continues to expand, it becomes increasingly essential to refine these models for real-life applicability in the vulnerability management process.

Vulnerability management, a critical aspect of cybersecurity, has evolved into an extensive field of research, focusing on the identification, evaluation, treatment, and reporting of security vulnerabilities

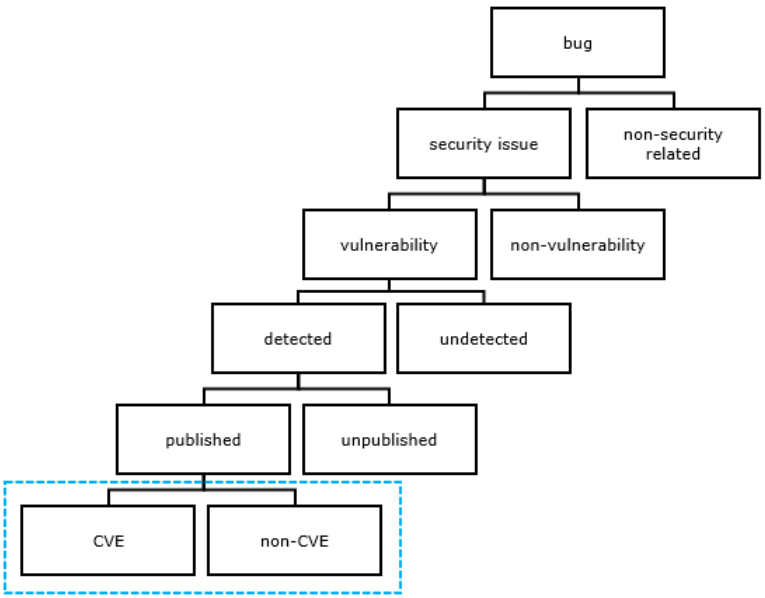

Figure 2.1: Classification of bugs and vulnerabilities, adapted from Schryen [3]. Dashed line indicating thesis focus area

in systems and networks. Scholars like Frei et al. [42] have delved into the quantitative analysis of vulnerability lifecycles, offering insights into the patterns of vulnerability disclosures and patch developments and emphasizing the importance of timely responses [42]. Meanwhile, studies by Allodi [43] have investigated the economic aspects of vulnerability management, revealing how market forces and financial incentives influence the disclosure and patching of vulnerabilities [43]. On the technical front, research by Joh and Malaiya [44] has contributed to the development of models for vulnerability discovery from code, emphasizing the probabilistic nature of finding vulnerabilities and the factors that affect this process [44].

2.2. Vulnerability Detection from Codebase

Li et al. [45] explore the use of deep learning to automate the process of software vulnerability detection from code bases, traditionally reliant on manual feature definition by human experts. Specifically, they developed a system for vulnerability identification titled VulDeePecker, which utilizes a deep learning architecture based on Bidirectional Long Short-Term Memory (BLSTM) networks, effective in processing sequences and capturing long-range dependencies [45]. This choice is strategic for handling the sequential nature of code, where the context of a line can be influenced by both preceding and succeeding lines. VulDeePecker introduces ”code gadgets,” semantically related segments of code, vectorized to serve as inputs for the deep learning model. To evaluate VulDeePecker, the authors created the first dataset designed specifically for deep learning-based vulnerability detection, which includes thousands of labeled code gadgets from real-world software systems. The results demonstrated that VulDeePecker significantly outperforms traditional vulnerability detection methods, particularly in reducing false negatives—crucial for systems where high false negatives can leave systems at risk of undetected threats. The system’s performance metrics are notably strong, with VulDeePecker achieving fewer false negatives and reasonable false positives compared to other approaches. The deep learning model’s ability to reduce the false negative rate significantly improves over traditional methods that rely heavily on human-curated features and heuristic analysis. However, while the results are promising, the study acknowledges several limitations that could affect the generalization and effectiveness of the system. One major limitation is the model’s dependency on the quality and representativeness of the training data. Since the performance of deep learning models is heavily reliant on the data used for training, any lack of diversity or comprehensiveness in the dataset can limit the model’s ability to generalize to new, unseen code bases or different types of software systems. Additionally, while VulDeePecker automates the feature extraction process, the initial setup and tuning of the model require substantial

expert knowledge, which could limit its applicability in environments with limited access to deep learning expertise.

Similarly, the paper ”Security Vulnerability Detection Using Deep Learning Natural Language Processing” by Noah Ziems and Shaoen Wu [46] presents an innovative approach to identifying security vulnerabilities in software using advanced natural language processing models. The authors propose a method where source code is treated as text, leveraging recent breakthroughs in deep learning and transfer learning to automate the detection of software vulnerabilities. Their dataset, derived from the NIST NVD/SARD databases, comprises over 100,000 files in the C programming language, highlighting 123 types of vulnerabilities. The study showcases the effectiveness of transformer models, particularly Bidirectional Encoder Representations from Transformers (BERT), achieving a remarkable accuracy rate of over 93% in identifying security threats. This high performance underlines the potential of deep learning models to revolutionize traditional vulnerability detection methods. The authors explore the challenges associated with detecting security vulnerabilities during software development, where traditional code analysis methods often fall short due to the complex nature and diversity of security threats. By framing software vulnerability detection as an NLP problem, the paper emphasizes the structural and patterned nature of code vulnerabilities, which can be effectively recognized and classified using deep learning models. The approach not only addresses the limitations of conventional error detection techniques but also introduces a scalable and efficient solution for real-time vulnerability identification.

2.3. Security Issue Classification

Vulnerability detection has also been explored from the issue tracker point of view. Gegick et al. [47] explore the challenge of accurately identifying security bug reports (SBRs) within large bug databases using text mining techniques. The authors develop a method that leverages natural language processing to analyze the descriptions in bug reports to distinguish between security and non-security-related issues. The approach involves training a statistical model on previously labeled data to classify and reclassify bug reports, namely, SAS Text Miner [47]. This model is particularly aimed at identifying SBRs that were incorrectly labeled as non-security bug reports (NSBRs) from Bugzilla by reporters who might not possess adequate security domain knowledge. For their study, the researchers utilized a dataset comprising bug reports from Cisco’s large software systems, which include over ten million source lines of code. The dataset included both security and non-security bug reports collected over several years. The evaluation metrics used to assess the model’s performance include precision, recall, and the percentage of correctly identified SBRs. The results were promising, showing that the model correctly identified 78% of the mislabeled SBRs with a probability of at least 0.98.

In 2017 Peters et al. [48] presented FARSEC, a novel framework aimed at improving the prediction and classification of security-related bug reports in software systems [48]. The challenge addressed by the paper is the difficulty in accurately identifying SBRs due to the overwhelming presence of non-security bug reports and the frequent occurrence of security-related keywords in both types of reports, leading to misclassification. FARSEC proposes a method to filter out NSBRs containing security-related keywords and rank the remaining bug reports to prioritize those most likely to be SBRs. The paper demonstrates that FARSEC enhances the performance of text-based prediction models in correctly identifying SBRs, through a series of experiments involving 45,940 bug reports from Chromium and four Apache projects. By effectively reducing the presence of misleading security-related keywords in NSBRs, FARSEC was able to decrease the number of misclassified SBRs by 38%. The paper’s methodology involves using machine learning algorithms combined with a filtering and ranking system that focuses on the textual content of bug reports. This approach not only addresses the issue of class imbalance (where the number of NSBRs significantly exceeds the number of SBRs) but also tackles the problem of ”security crosswords,” where the same security-related keywords appear in both SBRs and NSBRs.

Building on FARSEC, Zheng et.al [49] introduced in 2021 an improved method for predicting security bug reports utilizing domain knowledge. This method enriches supervised machine learning with security-specific insights from the Common Weakness Enumeration (CWE) and Common Vulnerabilities Exposure databases. The methodology involves generating matrices from bug reports using keywords rooted in the authoritative sources of CWE and CVE. These matrices serve as input for machine learning models to classify and predict SBRs. Specifically, the datasets introduced by Wu et al. [21] have been tested using the improved FARSEC model with the previously mentioned methodology [49].

The performance of the new method was compared directly to FARSEC, which generally relies on identifying security-related keywords in bug reports and filtering based on these keywords without the nuanced integration of domain-specific knowledge from sources like CWE and CVE. In contrast, the method proposed by Zheng et al. enriches the machine learning process with targeted domain knowledge, leading to more accurate predictions. The results of the study demonstrate a significant improvement in SBR prediction, with an average increase of 25% in the f1 score compared to the baseline method.

Additionally, recent studies have begun exploring the impact of machine learning (ML) in vulnerability management, suggesting that AI-driven approaches could revolutionize the way vulnerabilities are detected and mitigated [1], [50]. In 2017, Yaqin Zhou and Asankhaya Sharma [51] discussed an efficient, automatic vulnerability identification system aimed at open-source libraries. This system leverages natural language processing and machine learning techniques to analyze commit messages and bug reports from GitHub, JIRA, and Bugzilla, identifying vulnerabilities in real-time [51]. The primary motivation behind this work is the increasing reliance on open-source libraries in software development, which comes with the risk of using libraries that might contain unreported vulnerabilities. The paper highlights that a significant portion of vulnerabilities in open-source libraries are not publicly disclosed, thus putting software at risk of being hacked. The proposed solution is a K-fold stacking classifier that aggregates multiple individual classifiers to identify vulnerability-related commits and bug reports. This approach addresses the challenge posed by the highly imbalanced nature of available data, where vulnerability-related instances are scarce. The paper reports an improvement in precision by 54.55% over the state-of-the-art SVM-based classifier for commit messages, maintaining the same recall rate. For bug reports, the system achieves a precision of 0.70 and a recall rate of 0.71, indicating its effectiveness. Moreover, the system was deployed in production at SourceClear, where over three months, it demonstrated a precision of 0.83 and a recall rate of 0.74, detecting 349 hidden vulnerabilities [51]. The methodology involved collecting a vast amount of data, filtering out irrelevant information through regular expressions, and using word embeddings for feature representation. The classifier's design incorporates a probability-based classification to flexibly balance precision and recall, addressing the challenge of working with imbalanced datasets.

In 2023, Sultan S. Alqahtani [52] delves into leveraging FastText for classifying security bug reports (SBRs) in software development [52]. It highlights the importance of timely identification of SBRs to prevent exploits before public disclosure. The approach utilizes FastText, a machine learning model, for classifying bug reports as SBRs or non-SBRs, based on the public dataset of 45,940 bug reports from five open-source projects introduced by Wu et al. [21]. The results showcase FastText's efficacy, achieving an average F1 score of 0.81 for SBR identification and an average F1 score of 0.65 in cross-project validation, indicating its moderate generalizability. While the study presents significant findings, it relies on FastText, a method that, despite its efficiency and effectiveness, is considered less sophisticated compared to recent advancements in NLP technologies, particularly transformers. Transformers have revolutionized NLP tasks due to their ability to capture contextual relationships in text through mechanisms like self-attention, offering improved accuracy in text classification tasks. The transformer, introduced by Vaswani et al. [53], exchanges traditional recurrent models in favor of a purely attention-based mechanism, which allows it to model dependencies without considering their distance within the sequence, thus improving computational efficiency and model performance on tasks like machine translation [53]. Exploring transformers for classifying SBRs could potentially enhance model performance, providing deeper insights into the nuances of security-related reports and improving the detection of SBRs across diverse software projects. By leveraging the transformer's ability to efficiently handle long-range dependencies [53], future exploration of transformers in this domain could lead to more robust, context-aware classification systems, further advancing the field of software security.

In 2024, Meher et al. [54] present a novel deep-learning approach to classify software bugs into eight predefined categories: database, enhancement, infrastructure, logic, networking, performance, security, and usability. The motivation behind this research is to enhance the bug triage process, code inspection, and repair activities through accurate bug classification, leveraging the expressive power of deep learning. The authors first construct a bug taxonomy based on a set of keywords characterizing each of the eight bug classes. This is achieved through an iterative process of content analysis on bug summaries. Approximately 1.36 million software bug resolution reports were heuristically annotated

using an earth-mover distance technique based on the taxonomy's keywords. This step provides a ground truth for training deep learning models. A vanilla transformers-based classification technique and three transformer models, namely BERT, CoDeBERT, and DistilBERT, are employed to classify the annotated bugs. These models leverage self-attention mechanisms for improved classification accuracy. Experimental results on a curated dataset reveal that the proposed technique achieved a mean F1-Score of 84.78% and a mean macro-average ROC of 98.25%, outperforming existing techniques by an average of 16.88% in terms of F1-Score [54]. This significant improvement underscores the effectiveness of employing deep learning and self-attention mechanisms in software bug classification. The research not only advances the state of automated bug classification but also contributes a large annotated bug report dataset for future research.

2.4. Datasets for Security Issue Classification

Looking at the datasets used for the task of security issue classification, it can be observed that multiple authors make use of the Chromium dataset. By applying backward and forward snowballing on the paper of Wu et al. [21], which curated an initial version of the Chromium dataset introduced at the Mining Software Repository Challenge [55], the proposed models with the attached performance metrics from Table 2.1 have been identified. The Chromium dataset is seen across authors as a reliable, trust-worthy dataset to be used in the training of models, containing more data than the other available datasets [52, 49].

Model	Precision	Recall	f1 score
FARSEC[3] [21]	0.66	0.95	0.78
Random Forest (RF)[3] [21]	0.92	0.72	0.81
Domain knowledge RF[3] [49]	0.91	0.74	0.81
Active learning[3] [56]	[1]	[1]	0.87
fassttext[2] [52]	0.94	0.94	0.94
Random Forest (RF)[2] [57]	0.97	0.99	0.98

Table 2.1: Comparison of model performance metrics on the Chromium dataset curated by Wu et al. [21]

2.5. Challenges in Automatic Issue Classification

Data quality

One of the major challenges hindering the effectiveness of automatic methods for classifying security issues is the quality of the data used. As highlighted in several studies, poor data quality can significantly impact the performance of these models [45],[21]. This issue primarily arises because machine learning algorithms depend heavily on large, diverse, and accurately labeled datasets to learn from. In the context of security vulnerabilities, these datasets are often of reduced quality due to the presence of noisy labels, which contribute to decreased model accuracy and misclassifications.

Croft et al. [58] present a critical examination of the impact of data quality on software vulnerability prediction models. The authors identify and explore five inherent data quality attributes—accuracy, uniqueness, consistency, completeness, and currentness—across four publicly available, state-of-the-art software vulnerability datasets: Big-Vul, Devign, D2A, and Juliet [58]. The study found a significant proportion of vulnerability labels to be inaccurate, ranging from 20-71% across the datasets. Inaccurate labels can mislead models into learning incorrect patterns, thus impairing their ability to accurately predict vulnerabilities. This results in models that are either ineffective in identifying real vulnerabilities or that produce a high rate of false positives. Furthermore, duplication rates were observed to be between 17-99% in the datasets. Duplicate entries can lead to overfitting, where models perform exceptionally well on the training data but fail to generalize to unseen data. This can also inflate benchmark performance, giving a misleading representation of a model's true predictive capabilities. Lastly, some data points were found to have conflicting labels, causing confusion during model training. Inconsistent

[1] Metrics are not used by the author

[2] Models are trained on the Mining Software Repository Challenge Chromium version [55] (see Section 3.2.2)

[3] Models are trained on the Chromium dataset curated by Wu et al. [21] (see Section 3.2.2)

labels prevent models from learning clear distinctions between vulnerable and non-vulnerable code, leading to decreased model performance.

On the problem of label correctness, Wu et al. [21] present a comprehensive examination of the impact of label correctness in datasets used for SBR prediction. Focusing on five publicly available datasets, namely Chromium, Ambari, Camel, Derby and Wicket, the authors uncover a significant amount of mislabeling errors, where many instances classified as non-SBRs were, in fact, SBRs [21]. This mislabeling leads to poor performance in SBR prediction models, as demonstrated by the performance metrics of recent studies [48],[59]. To be more precise, a team of security experts analyzed and corrected the labels in these datasets and managed to identify 749 instances that were wrongly labeled as NSBRs, significantly enhancing the label correctness of the datasets. Subsequent evaluation of classification models on both the original (noisy) and corrected (clean) datasets revealed a notable improvement in model performance across various metrics, including Recall, Precision, F1-score, and G-measure, when using the corrected datasets. It is important to note that for the Chromium project, Wu et al. have cleaned the project bug tracker and found 808 security issues, compared to previously 192 that were identified. This is a significant increase in the number of security issues and allows for a larger database that can be used for training of classifiers. For our research, both versions of the database of Wu et al. on the Chromium project will be used to fine-tune the transformers for security issue identification and demonstrate the impact of data quality on model performance.

Data imbalance

Data imbalance in classification problems significantly impacts the performance of classifiers. When dataset classes are not approximately equally represented, most classifiers tend to exhibit a bias toward the majority class, leading to poor identification and prediction of the minority class. This problem is particularly critical in datasets where the minority class is of significant interest, such as in medical diagnostics or fraud detection. Traditional evaluation metrics like accuracy become misleading in these scenarios because a high accuracy rate might only reflect the predominance of the majority class rather than the effectiveness of the classifier in identifying the critical minority class. For instance, in a dataset where 95% of the data belongs to one class, a classifier that always predicts the majority class would achieve 95% accuracy, completely failing to identify the minority class.

The same situation also applies to the identification of security issues. Looking for example at the cleaned Chromium dataset by Wu et al. [21], the minority class is represented by the security issues and accounts for 1.92% of the data entries. Therefore, training a model on such an imbalanced dataset can lead to poor identification of security issues. To address the imbalance issue in data sets, several techniques can be employed. The Synthetic Minority Over-sampling Technique (SMOTE) is one such method, known for its simplicity and robustness in various applications. Unlike simple over-sampling, which can lead to overfitting and does not bring new information in the learning process, SMOTE enhances the representation of the minority class by creating synthetic examples through interpolating between existing minority class examples and their nearest neighbors. This helps broaden and generalize the decision boundaries for the minority class, effectively reducing the bias towards the majority class and increasing classifier sensitivity to minority examples [60]. SMOTE has proven effective in several domains, including bioinformatics, software engineering, and social media, significantly contributing to new supervised learning paradigms such as multi-label and incremental learning [60]. Another technique is under-sampling the majority class, which reduces the number of majority class entries to achieve a more balanced class distribution. Combining SMOTE with under-sampling has been empirically shown to outperform traditional resampling techniques and modifications of class distribution through algorithmic cost adjustments, particularly in improving the area under the ROC curve (AUC) [61].

The experimental results presented by Chawla et al. [61] in their paper indicate that SMOTE, particularly when used in conjunction with under-sampling of the majority class, effectively shifts the classifier's decision boundary towards a more balanced consideration of both classes. This shift results in improved performance metrics, especially in the recognition of the minority class without a substantial increase in the error rate for the majority class [61]. The approach has shown to be beneficial across various datasets characterized by significant class imbalances, making it a valuable technique for fields where the detection of rare but critical events is essential.

In the realm of software security, the paper by Tantithamthavorn et al. [62] investigates the effects of various class rebalancing techniques on defect prediction models in software engineering, focusing on both performance metrics and model interpretability across 101 datasets. Class imbalance, where the number of defective modules is substantially lower than non-defective ones, often biases defect prediction models towards predicting the majority class. This imbalance undermines the effectiveness of models by misrepresenting the true likelihood of defects. To counteract this, class rebalancing techniques such as oversampling through SMOTE and undersampling are applied by the author. These techniques adjust the training dataset to have a more balanced representation of both defective and non-defective classes. The study's findings illustrate that the impact of these rebalancing techniques varies with the performance metrics considered. For instance, while techniques like SMOTE can significantly enhance recall, they can concurrently lead to a decrease in Precision. This decrease in Precision results from the model predicting more false positives, where non-defective modules are incorrectly identified as defective. The balance between improving Recall without excessively sacrificing Precision is crucial and depends on the specific requirements and tolerance for errors in a given application. The paper also highlights the nuanced effects of rebalancing on model interpretability. Rebalancing techniques modify the decision boundaries of models, which can lead to changes in how features are weighted and interpreted. Such changes can obscure the understanding of which features are most predictive of defects, complicating efforts to derive actionable insights from the models. For instance, if a model initially identifies code complexity as a key predictor of defects, rebalancing might shift focus to other less relevant features, misleading developers about the critical areas to address for quality improvements.

Important to note is the use of thresholds in the evaluation of the models. In classification models such as those used in defect prediction, a "threshold" is a cutoff value that determines how the model's output—a probability score indicating the likelihood of an instance belonging to a particular class—is converted into class labels. For example, in binary classification like defect prediction, if the output probability exceeds a common threshold of 0.5, the module might be labeled as 'defective'; otherwise, it's considered 'non-defective'. Adjusting the threshold can significantly alter the balance between sensitivity (true positive rate) and specificity (true negative rate), impacting key performance metrics such as Precision and Recall (for more details on performance metrics, see section 4.1). Therefore, the threshold setting directly influences these metrics, where increasing sensitivity can increase false positives and vice versa. In practice, the appropriate threshold depends on specific application needs; for example, in software defect prediction, a lower threshold might be used to ensure high recall, accepting more false positives to avoid missing actual defects. Conversely, higher thresholds might be used where false positives carry high costs. For the context of this research, thresholds are important, as they can help increase the precision of models and lower the effort required from security researchers to validate predicted issues.

In summary, while class rebalancing techniques can significantly enhance the performance of defect prediction models by addressing class imbalance, their implementation must be carefully managed to maintain an optimal balance between different performance metrics and to preserve the interpretability of the model [62]. This study provides valuable insights into the trade-offs involved in using these techniques, as well as the use of thresholds to increase model performance metrics.

On the topic of security issue classification, Liao & Zhang [63] present a novel approach to addressing the challenge of data imbalance in SBR identification. The primary innovation of the method they introduce, SEDAC, lies in its use of a Conditional Variational Autoencoder (CVAE) to generate synthetic bug report vectors that help balance the dataset between security and non-security bug reports. The authors identify two main challenges in the existing methods of SBR identification: the lack of consideration for long-distance contextual information and the inability to generate a completely balanced dataset. To address these issues, SEDAC utilizes distilBERT to transform individual bug reports into vector representations that capture richer contextual relationships. Following this, a CVAE model is employed to generate additional synthetic vectors corresponding to SBRs, effectively balancing the number of SBRs with NSBRs. The effectiveness of SEDAC was tested on a substantial dataset comprising 45,940 bug reports from the Chromium project and four Apache projects, the same dataset used by Peters et al. [48]. The results demonstrated significant improvements in the g-measure compared to baseline models, with enhancements ranging from 14.24% to 50.10% [63]. This indicates that SEDAC not only addresses the imbalance issue but does so in a way that markedly boosts the identification

accuracy of SBRs.

Lastly, França et al. [64] explore the efficacy of various data rebalancing techniques for identifying security issues in bug reports. The study utilizes 13 different rebalancing techniques on a dataset of security bug reports from the Ubuntu operating system. This dataset is notably unbalanced, with security-related bugs forming a minority. The authors find that oversampling methods, particularly SVMSMOTE (Support Vector Machine Synthetic Minority Over-sampling Technique) and Random Undersampling, outperform other techniques in their effectiveness [64]. Oversampling is generally more favorable compared to undersampling due to its ability to enrich the dataset without losing valuable information from the majority class. The paper also tests the impact of varying proportions of the minority class on the performance of these methods, concluding that changes in these proportions do not significantly affect the results between the best-performing methods.

Cross-project Application

Cross-project prediction involves using a model trained on one project (or dataset) to predict outcomes on a different project. This approach is critical for practical applications where models need to perform well across various codebases without the necessity for retraining on each new project. The paper by Chakraborty et al. [25] systematically tests the generalizability of four state-of-the-art neural network architectures by applying them to datasets they were not trained on in the context of vulnerability detection. The authors begin by outlining the typical high performance of state-of-the-art neural network architectures within the datasets they were trained on. These architectures, which have been previously celebrated for their high accuracies—sometimes as much as 97%—are scrutinized for their ability to maintain such performance when applied to unseen, external datasets. The study's results are illuminating: all tested models exhibit a steep decline in accuracy when they are applied to new datasets, highlighting a significant generalization issue. To investigate the underlying causes of this generalization problem, the paper discusses the role of dataset biases that affect machine learning models. These biases are often introduced during the data collection, curation, and labeling processes. For instance, datasets might be biased towards certain types of vulnerabilities or specific coding styles that are not universally applicable. Such biases lead models to learn idiosyncratic features that do not necessarily correlate with true vulnerability characteristics but rather with the specific context of the training data. In response to these findings, the authors compile a new dataset from real-world projects—namely Chromium and Debian—which is carefully balanced and more representative of real-world conditions than the previously used datasets. This dataset is free from the typical biases that skew model training in other datasets. However, when the neural networks trained on traditional datasets are tested against this new dataset, they only achieve around 50% accuracy, significantly lower than the accuracies reported on their original datasets [25]. This stark difference demonstrates the models' inability to adapt to new environments and data characteristics, a critical flaw for practical applications where software varies greatly across projects. The paper concludes that current machine learning models for vulnerability detection are inadequate for cross-project applications due to their limited ability to generalize beyond the data they were trained on. This limitation stems largely from dataset-specific biases and the lack of robustness in the models to adapt to diverse, real-world data. The findings suggest a pressing need for the development of new models and training approaches that prioritize generalizability and robustness, potentially through more diverse training scenarios and the incorporation of techniques that minimize the impact of data biases.

The authors have discussed and analyzed a very pertinent question of cross-project implementation of the models. However, the focus of the paper, as discussed, centered around the generalization of neural network architectures for detecting software vulnerabilities from codebase and their performance on cross-project datasets. In this research, we will focus on using more advanced deep-learning algorithms, specifically transformers, to predict security issues based on text from the bug reports.

2.6. Coordination Efforts

In order to deal with the above-discussed vulnerabilities, enterprises, and organizations rely on vulnerability databases. The CVE program aims to standardize the identification of vulnerabilities across various platforms and tools by providing a unique CVE ID for every recognized vulnerability. CVE IDs are attributed by a diverse number of organizations and researchers named CVE Numbering Authorities

(CNAs). Lin et al. [14] have investigated the practices used by the CNAs to coordinate on vulnerability fix releases, providing insight into how security vulnerabilities are prioritized, attributed CVEs numbers and patched. Furthermore, CVE vulnerability descriptions are enhanced by NVD analysts (see figure 2.2) with additional information, such as the Common Vulnerability Scoring System value (CVSS) and affected products and versions [14]. The CVSS is a quantitative metric that uses formulas to return the severity of a vulnerability [65].

In this section, we have briefly covered important pieces of work from the research field of vulnerability management, which will be the cornerstone of the proposed research. Finally, the stakeholders involved in the vulnerability coordination can be presented. The vulnerability coordination follows the process depicted in figure 2.2. The main stakeholders involved are the following:

CVE Numbering Authorities (CNAs): These organizations, companies, and researchers act as security experts within their specific areas of coverage. They evaluate vulnerabilities, assign CVE IDs, and control the disclosure process.

Software Vendors: These entities are often affected by vulnerabilities and are responsible for developing and releasing fixes. In many cases, the CNAs and software vendors are the same entities.

Discoverers: Individuals or organizations that find vulnerabilities. They report these to affected software vendors and coordinate with them for mitigation plans.

National Vulnerability Database (NVD) Analysts: They analyze CVE records and update information such as CVSS scores and affected products.

Users: The end-users of the software, who become aware of vulnerabilities from CVE records and rely on software vendors for fixes.

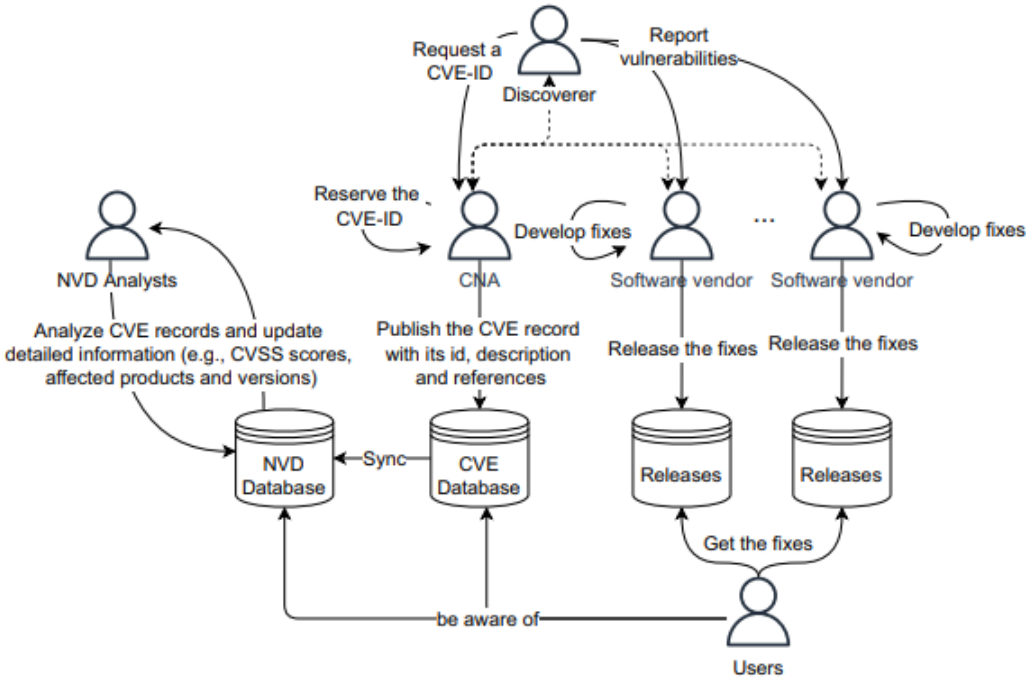

Figure 2.2: Overview of multi-party coordination on fixing vulnerabilities, as depicted in Lin et al. [14]

The stakeholders identified are integral participants in a process known as Coordinated Vulnerability Disclosure (CVD). CVD is a critical strategy that many organizations employ to manage the disclosure of security vulnerabilities effectively. This process involves multiple stakeholders including security researchers, project maintainers, and vendors who collaborate to identify, fix, and publicly disclose vulnerabilities in a way that minimizes the risk to users. The interaction among these parties is vital for the success of CVD, yet it is fraught with challenges that can arise from conflicting goals or mis-

understandings. The CERT Guide to CVD [13] outlines a comprehensive framework for how various stakeholders—ranging from vulnerability discoverers to vendors and coordinating bodies—should interact to ensure vulnerabilities are disclosed responsibly and efficiently. The guide emphasizes the importance of a clear communication channel among all parties involved. It suggests that the establishment of pre-defined roles and responsibilities is crucial for maintaining an orderly flow of information. Walshe & Simson's [66] research provides a critical examination of the challenges facing these programs. High volumes of low-quality reports, for instance, are identified as a significant burden. These reports can consume substantial resources, slowing down the response to more critical vulnerabilities and diminishing the overall effectiveness of vulnerability management programs. The research also points to the distrust between organizations and hackers as a substantial barrier. This mistrust can deter organizations from engaging with external security researchers, potentially leading to delays in identifying and resolving security issues. Furthermore, the paper discusses the operational challenges, such as the discrepancies in vulnerability severity assessments and delays in patch deployment, which can significantly impact the timeliness and effectiveness of the CVD process. Moreover, the lack of skilled personnel to manage the disclosure process is a recurring theme in Walshe's study [66]. The paper emphasizes that the effectiveness of CVD programs heavily relies on the expertise of those who manage them. Skilled personnel can better triage reports, assess the severity of vulnerabilities accurately, and communicate effectively with all stakeholders involved.

To conclude, the research on CVD illustrates a complex ecosystem where the interplay of various factors determines the success of vulnerability disclosure programs. While the CERT guide provides a structured approach to managing disclosures, research findings highlight the practical challenges that often hinder these theoretical frameworks from functioning optimally [66]. Together, these works suggest that for CVD efforts to be more effective, there must be an organizational commitment to security, a strategic emphasis on skilled personnel, and a cultural shift towards trusting and collaborating with the external security research community. Incorporating automated vulnerability detection tools could significantly enhance the efficiency of CVD processes by enabling quicker responses to high-risk publicly disclosed vulnerabilities.

3

Methodology

The main research question to be answered in this research is: *What is the extent of vulnerabilities in open-source projects on GitHub that do not have CVE identification numbers?* The methodology employed to address the research question is illustrated in Figure 3.1 and consists of the following blocks. First, the scope of the study is defined, focusing on C++ projects to understand their unique security challenges. Next, a security issue identification system is developed through model selection, training dataset selection, model fine-tuning, and threshold modification. This system is then applied to different projects to predict security issues. Finally, the process concludes with expert validation and results analysis, involving security experts validating security issues into security vulnerabilities and estimating the CVSS 3.1 score to assess the depth of the uncovered vulnerabilities. The remainder of this chapter is structured following the methodology steps and goes into more detail in terms of research design choices.

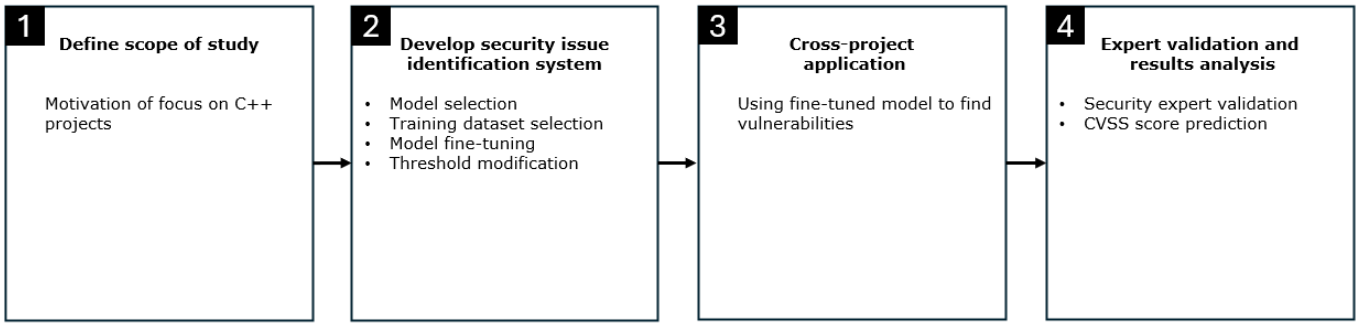

Figure 3.1: Research methodology steps

3.1. Scope of the Study

The sustained prominence of C and C++ in the programming world underscores their significance as target languages for software development and security analysis. According to the TIOBE Index, which assesses the popularity of programming languages, C and C++ constantly rank number 2 and 3 in the world, after Python [67]. The TIOBE index measures the popularity of programming languages based on the number of skilled engineers, courses, and third-party vendors worldwide using the programming language [67]. The persistent ranking of C and C++ near the top of this index not only reflects their enduring relevance but also signifies the extensive base of code written in these languages that requires ongoing maintenance and security evaluation. Popular projects such as TensorFlow [10], React Native [68] and OpenCV [69] are built on C++ and used throughout industries. This widespread application further elevates the importance of addressing and understanding its vulnerabilities as they can have wide-reaching impacts. Therefore, this study aims to investigate security vulnerabilities associated

specifically with C++ projects.

Unfortunately, the C++ programming language has a propensity for security vulnerabilities, notably memory-related vulnerabilities [70]. C++ is a powerful language offering fine-grained memory management capabilities; however, these same features often result in complex memory management bugs and security flaws if not handled appropriately. The decision to focus on C++ is substantiated by the language's history and structure, which inherently predisposes it to these issues. C++ is considered to be a weakly typed language because a memory region of a certain type can be treated differently using pointer manipulation [71]. C++ allows for direct memory management, which while providing performance benefits, significantly increases the likelihood of memory leaks—where memory which is no longer needed is not released [71]. Memory leaks are a predominant issue in C++, as they can lead to reduced performance and system stability. According to research by Lacroix et al. [72], C++ applications frequently exhibit security vulnerabilities related to memory management such as buffer overflows and improper memory operations, which are less prevalent in languages that manage memory automatically. Moreover, a comparative study by Ray et al. [73] on software languages, which analyzed GitHub projects, highlighted that C++ is more prone to errors and vulnerabilities compared to other programming languages. This study found that C++ codebases have a higher rate of security issues per line of code, underscoring the language's susceptibility to potentially exploitable vulnerabilities. The prevalence of vulnerabilities in C++ is also linked to its widespread use in applications requiring high-performance computing, such as game development, system programming, and real-time simulation.

3.2. Security Issue Identification System

3.2.1. Model Selection

In the field of software security, the classification of security issues is crucial for identifying and mitigating vulnerabilities that could potentially be exploited. As outlined in the literature review, methods for classifying security issues are diverse, spanning from simple heuristic-based approaches to sophisticated deep-learning models. This spectrum of methods reflects the evolution of technology and methodologies in the field of cybersecurity. In this research, the task of the model would be to classify an issue either as security or non-security-related. Based on labeled datasets, methods for this task have been previously explored, such as text-mining [47], Random Forest [48], K-fold stacking [51] and FastText [52], as outlined in Chapter 2. Recently, deep learning came to the attention of researchers for issue classification, with Meher et al. [54] proposing a transformer-based model for the classification of issues in 8 categories.

Transformers are a groundbreaking model in the field of deep learning, primarily designed to handle sequential data, and have been instrumental in advancing natural language processing and other areas of artificial intelligence. Introduced by Vaswani et al. [53] in the paper titled "Attention is All You Need," transformers eliminate the need for recurrence in network architecture by utilizing a self-attention mechanism [53]. This mechanism processes an entire sequence of data simultaneously, allowing for significantly increased parallelism compared to architectures like Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU).

Following their introduction, transformers have been rapidly adopted for a wide array of applications, demonstrating remarkable effectiveness. They form the backbone of models like BERT [18], which has significantly advanced the state-of-the-art in tasks such as question answering and language inference. The versatility of transformers is further evidenced by their adaptations in non-NLP tasks, such as image processing and vision tasks, as demonstrated in the Vision Transformer (ViT) model by [74].

Moreover, the scalability of transformers has been explored in large-scale models like GPT-3 [75], which utilizes a massive transformer-based architecture with hundreds of billions of parameters. The performance of GPT-3 highlights the transformer's capability to handle and generalize from vast amounts of data, setting new benchmarks across various linguistic tasks.

For this research, five different transformer models have been employed across two approaches. The first approach was fine-tuning transformers, for which we have leveraged the BERT-based models: Bidirectional Encoder Representations from Transformers (BERT) [18], Robustly Optimized BERT Approach (RoBERTa) [19], and Decoding-enhanced BERT with Disentangled Attention (DeBERTa) [20]. The second approach involved few-shot prompting two general purpose models: Gemma [22] and

Mistral [23].

BERT based models

BERT is a groundbreaking approach in natural language processing introduced by Devlin et al. [18] in their 2018 paper. It leverages transformer architecture to pre-train deep bidirectional representations from unlabeled text by conditioning on both left and right context. This method enables the model to be fine-tuned with just one additional output layer to achieve state-of-the-art results on a wide range of NLP tasks, such as question answering, language inference and text sequnce classification. BERT's effectiveness is primarily due to its ability to capture nuanced language contexts, setting new performance benchmarks across various NLP benchmarks.

RoBERTa [19] is a model based on Google's BERT model previously described. It builds on BERT and modifies key hyperparameters, removing the next-sentence pretraining objective and training with much larger mini-batches and learning rates. This allowed the model to obtain better results and made it an overall more robust model for NLP tasks [19].

DeBERTaV3 [20] is the latest of the DeBERTa models and it improves the BERT and RoBERTa models using disentangled attention and enhanced mask decoder. With those two improvements, DeBERTa outperforms RoBERTa on a majority of Natural Language Understanding (NLU) tasks with 80GB training data [20].

General purpose models

Mistral [23] is a language model introduced in September 2023 by Mistral AI [23], a French company, and represents a major advance in large language model (LLM) capabilities.

Gemma [22] is a new family of open Large Language Models introduced by Google in February 2024 that stand out for their versatility and efficiency. Gemma outperforms Mistral 7B on most tasks and approaches the performance of the much larger LLama 2 70B model.

Compared to the BERT-based models, which required fine-tuning in order to prepare them for the text classification task, the models above have not been fine-tuned but few-shot prompted. Few-shot prompting is a method where a language model is given a small number of examples (shots) to help it understand a task it needs to perform. By showing the model just a few instances of the input-output format for the issue classification task, it can generalize and apply this pattern to new, unseen inputs. This technique leverages the model's pre-existing knowledge, gained during its training, to perform tasks without extensive fine-tuning or large datasets for each specific task.

The few-shot prompt consisted of six issue descriptions from the Chromium dataset, three security and three non-security, as well as the expected output. The few-shot prompt is presented below, where the issue descriptions between asterisk remain the same for each prediction, but the ”bug to classify” variable changes for each entry of the test dataset. At the end, the assistant (the LLM) answer is left empty, which is how the model knows when it is time to answer the prompt.

Few shot prompt: User: Given the following bug description from a C++ project, classify the bug as either ”security” or ”non-security”:*Description of non-security bug 1*

Assistant: non-security

User:*Description of security bug 2*

Assistant: security

User:*Description of security bug 3*

Assistant: non-security

User:*Description of security bug 4*

Assistant: security

User:*Description of security bug 5*

Assistant: non-security

User:*Description of security bug 6*

Assistant: security

User:*__bug to classify__*

Assistant:

3.2.2. Training Dataset

In this subsection, the datasets used in the study are introduced. For the determination of the issue classification feasibility into security and non-security issues, labeled datasets are required. As presented in Chapter 2, labeled datasets for security vulnerability identification from bug repositories are scarce, and often come with associated challenges, such as dataset size and quality. In the literature review, it was identified that multiple authors have explored the issue of vulnerability identification based on the publicly available Chromium dataset. Furthermore, the dataset has been curated by Wu et al. [21], increasing the quality of the assigned labels, which gives our thesis sufficient confidence to leverage the Chromium dataset. Furthermore, it is considered to be a dataset of good quality by multiple authors [52, 49, 56] and the results of previous authors can represent a good benchmark to compare the performance of the proposed models.

Dataset	SBR	NSBR
Chromium - small	192	41748
Chromium - large	808	41132

Table 3.1: Datasets used for model fine-tuning

For this reason, two versions of a publicly available dataset of the C++ based Chromium project have been used, as can be seen in table 3.1. The first version is shared by the 2011 mining challenge of the MSR conference [55], which contains 41940 bug reports, out of which only 192 or 0.46% are SBR. This version will be referred to as the ”small” Chromium dataset, as the SBR class is of reduced size. The second version of the dataset is presented by Wu et al. [21] and represents a curated version of the small Chromium dataset, as the number of SBR’s has increased to 808, or 1.93% of the total issues (see Section 2.5 on data imbalance). The dataset of the Chromium project has been used for both training and testing purposes [21]. Each row entry of the dataset has the column entries *bug id*, *bug description*, *date*, *security label*. The bug descriptions, which are identical for both datasets, have been collected from the period August 2008 to June 2010. For the work presented, only the *bug description* and *security label* have been used. In the bug description column, the bug report text is presented, while the security label contains binary values, with 1 if the bug is security-related or 0 if it is non-security related. Below an example of a security issue from the original dataset is presented.

Issue description: Issue 1643 : The memory leak in the root process ‹ Prev 5951 of 14144 Next › 5 problem? After several days of usage the root process allocated 1.6GiB of memory. And this process could not be restarted or selectively killed. ? The root process must not leak the memory or it should be easily restartable picking existing page processes after restart. Fro example page navigation could be done by the separate child process. That respawns if it dies. ? 1.6GiB is allocated.

Security label: 1

Because it is expected that a large part of the security vulnerabilities in C++ projects are related to memory issues, it may seem adequate to identify security vulnerabilities in software issue descriptions by simple text searches like ”memory” or ”memory leak,” however, the results from such methods are insufficient for comprehensive vulnerability detection, as depicted in table Table 3.2. For instance, using the large Chromium dataset as an example, searching for ”memory” in the issue descriptions yields 1077 hits, of which only 361 are related to confirmed security issues. Narrowing the search to ”memory leak” results in 230 hits, with 227 being relevant to security issues. Although ”memory leak” as a query term appears more precise, it still captures only a subset of potential vulnerabilities. Importantly, these queries overlook other types of vulnerabilities present in the dataset. In fact, focusing solely on ”memory” related terms excludes a significant number of other security issues, up to 808

vulnerabilities identified within the same Chromium dataset. This limitation underscores the necessity for more sophisticated approaches beyond keyword searching. Machine learning techniques, particularly those utilizing advanced models trained on security-specific datasets, provide a more effective solution. These techniques can discern subtle patterns and contexts indicative of a broader range of vulnerabilities, thereby enhancing the detection process significantly.

total issues	query	identified issues	identified security issues
41940	memory	1077	361
	memory leak	230	227

Table 3.2: Summary of issues identified in descriptions

Implications: A simple search query, while effective for identifying memory leaks in C++ projects, is insufficient for detecting the wide array of other types of vulnerabilities that may be present. This method, therefore, captures only a narrow aspect of the security challenges, overlooking a significant majority of potential threats. Consequently, it is evident that memory leaks represent just one facet of software vulnerabilities in C++ projects. To address this limitation, more sophisticated identification techniques are required. The deployment of machine learning models offers a promising alternative, providing a comprehensive and nuanced approach to vulnerability detection that can adapt to the complex nature of software security issues.

3.2.3. Model Fine-tuning

Transformers generally don't need complex text preprocessing like stop-word removal because they are pre-trained on extensive text datasets. However, it is important to remove any distinctive text, such as long URLs, that might bias the model. This also helps shorten the input token count, lightening the processing load for transformers. To the entries of the bug description column, basic text preprocessing steps have been applied. First, the beginning of the bug description, stating "Issue number" (in the example above Issue 31517") has been removed from all entries, as it was a recurring section. Then the steps of changing the text to lowercase, removing html and URLs, cleaning special characters, and removing white spaces were applied. Below is the issue presented in Section 3.2.2 after being run through the pre-processing steps.

Preprocessed issue: The memory leak in the root process lsaquo; Prev 5951 of 14144 Next rsaquo; 5 problem? After several days of usage the root process allocated 1.6GiB of memory. And this process could not be restarted or selectively killed. ? The root process must not leak the memory or it should be easily restartable picking existing page processes after restart. Fro example page navigation could be done by the separate child process. That respawns if it dies. ? 1.6GiB is allocated.

As previously mentioned, the small Chromium dataset contains roughly 0.5% security issues. This means that the large majority of entries are non-security related, and point to the problem of data imbalance presented in the literature review. The class of non-security issues is majoritarian, and the proposed transformer models are biased towards learning from the majority rather than from the minority. To solve this problem, standard dataset balancing techniques have been used on the small version of the Chromium dataset.

1. Data over-sampling: The security issues are relatively few (only 192). To increase the size of the minority class, over-sampling has been applied on the security entries using BERT-based text augmentation. The augmentation has been performed at a word level using the *nlpaug.augmenter.word* library and base BERT as a model. The augmenter modifies the text at the word level. This can involve substituting words with other words that have similar meanings within the same context. Because BERT understands the context around each word, the substitutions it makes are contextually appropriate, ensuring that the new text remains meaningful and relevant to the original context. The augmentation was performed once per security entry and this allowed for the doubling of security entries, from 192 to 384. Below is an example of the previous security issue, after it has been augmented.

Augmented issue: The memory might leak in the third root process & lsaquo ; prev 5951 of 14144 next & rsaquo ; 5 task problem? possibly after several days availability of usage the main root process

allocated 1. 6gib of memory. and this process could not be restarted easily or selectively killed.? the root process must not successfully leak the memory or it should be easily restartable picking existing page processes after restart. fro example page navigation could be done by the separate child process. that respawns if again it dies.? suppose 1. 6gib is allocated.

2. Data under-sampling: The non-security issues are dominant. To counter this bias, a re-sampling technique has been applied in order to balance the training dataset. Out of the non-security entries, only 384 random non-security entries have been chosen. This re-sampling ensures a balanced class distribution. Important to note is that the data over and under-sampling has been applied only to the training dataset. The evaluation of the models will be conducted on real-life examples, with the ratio of security to non-security bugs remaining at 0.5%.

For the large version of the Chromium dataset, there are 808 issues marked as security, which represents 1.93% of the total issues in the dataset. Because this class increased in size compared to the small version of the Chromium dataset, over-sampling will not be applied on the minority class. However, the dataset is still unbalanced, containing over 41.000 non-security issues. By applying under-sampling on the non-security class for the training dataset we are able to obtain a 1616-entry dataset, assumed to be sufficient for the task of security issue classification.

For model fine-tuning a notebook has been created which is attached in Appendix B. A high-level description of the model training and testing is presented below. The notebook is run on a NVIDIA T4 GPU provided by Google Colab. The T4 GPU offers 16 GB of GDDR6 memory and is optimized for both training and inference of machine learning models [76]. It has been observed that with the current configuration, the T4 is sufficient for the fine-tuning process. However, changes in configuration, such as the increase of the max length of sequences from 512 to 1024 can cause the notebook to crash due to the runtime not having enough memory allocated.

The training initiates by loading a pre-trained model from Huggingface (more details on models used in the research are available in Section 3.2.1). Alongside the model, a tokenizer compatible with the model is initialized. This tokenizer processes text to convert it into a format suitable for the model, handling tasks such as splitting the text into tokens, converting tokens to IDs, and padding or truncating sequences to a uniform length of 512 tokens. The dataset used for training and evaluation is loaded from a pre-processed CSV file. This dataset is then split into training and testing subsets with an 80-20 ratio, ensuring that both data subsets are representative yet distinct.

The training and testing data undergo a tokenization process, where the text data is batch-processed to transform each text entry into the input format required by the loaded model. This includes converting text into input IDs and attention masks, which help the model focus on relevant parts of the input data during training. These processed inputs are then set to the appropriate format for use with PyTorch, a deep-learning framework that facilitates model training.

For model training, a series of configurations are specified, including the number of training epochs, batch sizes, and evaluation strategies. Notably, the model uses gradient accumulation and half-precision (FP16) training, techniques that improve training efficiency and scalability. The training process is closely monitored with a custom metrics function that computes accuracy, precision, recall, and F1 score, alongside maintaining a confusion matrix for deeper performance analysis. The trainer, a component from Hugging Face's Transformers library, orchestrates the training and evaluation process. It leverages early stopping to prevent overfitting, ensuring the model does not continue to learn from the training data past the point of diminishing returns. The best model is saved and reloaded at the end of the training, based on accuracy—a critical factor for ensuring that the most effective model configuration is used for future predictions.

It is important to note that the 80-20 train-test split is done on the balanced dataset. For real-life scenario testing, a separate, non-colliding subset of the Chromium dataset has been used to understand model performance on a dataset with an SBR to NSBR ratio introduced in Section 3.2.2. Finally, the model's performance is evaluated on the unbalanced subset to assess its effectiveness in real-world scenarios. This training and evaluation process ensures that the model is robust and efficient, through training on a cleaned, balanced dataset and ready for deployment in real-life application, by being evaluated on a dataset with unbalanced classes. The configurations and model code is available in Appendix B.

3.2.4. Threshold Modification

An important aspect to highlight is the application of threshold modification to manipulate the precision-recall tradeoff. The precision-recall tradeoff plays a critical role across various fields that utilize classification models, such as fraud detection. This tradeoff navigates the balance between precision—the proportion of true positive results among all positive predictions—and recall—the proportion of true positive results among all actual positives. For additional details on performance metrics, including precision and recall, please refer to Section 4.1. This balance becomes especially crucial in scenarios where the consequences of false positives and false negatives are significant. For example, in our research, a model characterized by high recall but low precision can identify all security issues but will also generate numerous false positives.

An already explored domain of the precision-recall tradeoff is fraud detection where high precision ensures that actions taken based on detected fraud are justifiable and minimize disruptions to legitimate transactions, which is vital for maintaining customer trust and satisfaction. Conversely, high recall is essential to capture as many fraudulent activities as possible to mitigate financial loss. Fraud detection models are typically tailored to optimize either precision or recall based on specific business and operational priorities [77]. Adjusting these models according to the current risk assessment and tolerable levels of false positives or negatives is common practice [77]. For our model which aims to accurately classify security issues, given the limited availability of security experts, it is impractical to sift through a large number of false positives. This issue is also highlighted by Wu et al. [21], who noted that although the FARSEC model achieved a recall of 0.85 [48], the precision of 0.25 resulted in 5,388 false positives, rendering the deployment of such a tool impractical [21]. Thus, in the identification of security issues, the precision-recall tradeoff is pivotal. High precision decreases the number of false positives—erroneously flagged security issues, which could otherwise waste the time of experts during the validation process. High recall ensures that more actual security issues are detected, significantly reducing the risk of overlooked vulnerabilities that could be exploited. This tradeoff often involves deciding how much risk is acceptable in failing to detect a vulnerability versus the resources allocated to investigate false security alerts.

To address the challenges posed by the precision-recall tradeoff, we propose utilizing post-training threshold modification. After training the model on the Chromium database, we will plot the distribution of positives across the confidence range. The default threshold of 0.50 in binary classification tasks (security or non-security) will be adjusted to achieve acceptable performance metrics. Detailed investigations on threshold modification are presented in Section 4.2.3. By altering the threshold level, we can refine the predictions dataset to a manageable size, minimizing the inclusion of false positives while ensuring it captures most security issues. This approach allows the model to be effectively applied across projects, and vulnerabilities can be validated by experts.

3.3. Cross-project Application

One of the big challenges of security issue classification models is cross-project applicability. Although some authors have applied trained models on other projects in order to investigate if security issues can be predicted effectively [52], in the literature it is disputed that cross-project applicability of the models usually leads to a significant decrease in model performance [25]. Therefore, our study does not focus specifically on measuring the performance of the fine-tuned models cross-project for security issue identification, but aims to go one level deeper than security issues and look at security vulnerabilities. As previously mentioned, not all security issues are security vulnerabilities. Therefore, the predicted security issues by the model will be investigated by a security expert to determine the number of possible untracked security vulnerabilities.

Once a transformer model is fine-tuned, it can be utilized to predict security issues from the issue trackers of other GitHub projects. The process involves several steps. Firstly, issues are extracted from the desired GitHub repository, identified by its repository and project name (see Appendix A for code). Secondly, these issues undergo the same preprocessing steps as the training and testing data, excluding the data balancing part. Lastly, the fine-tuned model is employed to predict the security label for each preprocessed issue.

3.4. Expert Validation and Results Analysis

In order to answer the research question, the predictions presented in Section 3.3 require validation, in order to separate security vulnerabilities from other types of bugs. For the validation of the security findings, a security expert who is also part of a Certified Numbering Authority is asked to review the results of the model's predictions. The issues predicted by the model as security-related are presented to the expert, who will conduct a thorough investigation to determine whether each issue indeed constitutes a software vulnerability. Furthermore, the expert will assess and assign a severity score to each confirmed vulnerability based on the CVSS 3.1 metrics, providing a nuanced understanding of the potential impact and urgency of the vulnerabilities identified.

The Common Vulnerability Scoring System (CVSS) serves as an open framework designed to communicate the characteristics and impacts of software vulnerabilities [78]. Its main objective is to provide a standardized scoring system that allows IT professionals to assess the severity of vulnerabilities in a consistent, repeatable manner. CVSS is intended to help organizations prioritize their vulnerability management and remediation decisions based on the severity of vulnerabilities [79]. Developed by FIRST (Forum of Incident Response and Security Teams), CVSS has undergone several revisions to improve its accuracy and usability in various contexts. Version 3.1, the latest iteration, includes updates and clarifications to the scoring guidelines, aimed at increasing the precision and reproducibility of vulnerability scores across different organizations and technologies.

CVSS 3.1 includes three metric groups—Base, Temporal, and Environmental—each assessing different aspects of a vulnerability. The description of the metric groups below has been introduced by the Forum of Incident Response and Security Teams in 2021 [79]. For the estimation of the CVSS 3.1 score, the expert has based the results solely on the base metrics:

Base Metrics: These metrics form the core of the CVSS scoring system and are intended to capture the inherent qualities of a vulnerability that are constant across different user environments. These metrics are divided into two categories: Exploitability and Impact.

Exploitability

Attack Vector (AV): This metric assesses how the vulnerability is exploited. For example, can the attack occur over a network (N), or does it require physical access (P) to the vulnerable component?

Attack Complexity (AC): Reflects the conditions beyond the attacker's control that must exist to exploit the vulnerability. This could include factors like the configuration of the target system that might require advanced knowledge to exploit.

Privileges Required (PR): Indicates the level of privileges an attacker must possess before successfully exploiting the vulnerability, ranging from none to high.

User Interaction (UI): Determines whether the exploitation of the vulnerability requires actions from a user, such as clicking a malicious link.

Impact Metrics:

Scope (S): Captures whether a vulnerability impacts components beyond its own security authority. A scope change occurs if the impact of a vulnerability affects resources beyond the initially compromised component.

Confidentiality, Integrity, and Availability (C, I, A): These metrics evaluate the impact on the three key aspects of information security. Confidentiality measures the impact on the unauthorized disclosure of information, Integrity on unauthorized modification or destruction of information, and Availability on the disruption of access to or use of information or an information system.

The expert results will contain the CVSS 3.1 score as described by the above metrics. The detailed breakdown of the vulnerabilities will allow for an investigation to be made on the most common aspects of the vulnerabilities. Below is an example of a CVSS 3.1 score of 5.1

Vulnerabilty CVSS 3.1 score 5.1: AV:L/AC:H/PR:N/UI:N/S:U/C:N/I:N/A:H

4

Results

4.1. Performance Evaluation Metrics

Evaluating the performance of the employed models is crucial for several reasons. Firstly, it allows one to assess the model's effectiveness in making accurate predictions or classifications. Secondly, it helps in tailoring the finetuning process in order to increase the effectiveness of the classification. For this research, it is important to understand if the proposed models can be effective in finding security issues that can be subsequently validated by security experts to determine the number of present CVEs. Evaluating model performance with metrics tailored to the task at hand, such as accuracy, precision, recall, and f1 score will be of help to make informed decisions based on empirical evidence for predicting security issues in new projects. The metrics that have been used are based on the entries of the confusion matrix and are standard to use for vulnerability classification [39]. A confusion matrix is a table used to evaluate the performance of a classification model on a set of test data for which the true values are known. It cross-tabulates the actual and predicted classes, providing insight into the types of errors made by the model. The confusion matrix for a binary classification problem consists of four different outcomes: true positives, true negatives, false positives, and false negatives (see Table 4.1).

	Predicted	
	Positive	**Negative**
Actual Positive	True Positives (TP)	False Negatives (FN)
Actual Negative	False Positives (FP)	True Negatives (TN)

Table 4.1: Confusion Matrix

1. Accuracy

Explanation: Accuracy measures the proportion of total predictions that the model got right. It is a straightforward metric that provides a quick overview of the model's effectiveness across all categories.

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{4.1}$$

2. Precision

Explanation: Precision assesses the model's accuracy in predicting positive labels. It is crucial for scenarios where the cost of a false positive is high. For this research, the cost of false positives would refer to how much time and effort it takes a security researcher to validate a bug and determine if it is security or non-security-related.

$$\text{Precision} = \frac{\text{TP}}{\text{TP + FP}} \tag{4.2}$$

3. Recall

Explanation: Recall measures the model's ability to detect all relevant instances. It is particularly important in situations where missing a positive instance (a false negative) carries a significant penalty. In this research, recall is assumed to be very important, as one does not want to miss security-relevant issues from the issue tracker.

$$\text{Recall} = \frac{\text{TP}}{\text{TP + FN}} \tag{4.3}$$

4. F1 Score

Explanation: The f1 score is the harmonic mean of precision and recall. This metric is useful when you need to balance precision and recall, especially when the dataset is imbalanced.

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision + Recall}} \tag{4.4}$$

When choosing a model for vulnerability detection, one must consider the trade-offs between these metrics based on the cost of false positives versus false negatives. For instance, in security-sensitive contexts, a high recall might be preferred to ensure all potential vulnerabilities are identified, even at the cost of increased false positives (lower precision). However, if the cost of examining false positives is high, a balanced f1 score might be more desirable.

4.2. Performance Evaluation

The classification methods presented in Section 3.2.1 have been applied according to the methodology on the two Chromium datasets. Each of the BERT-based models has been trained a total of 10 times on the same dataset, and the average results of the training process are presented in Table 4.2. Conducting the training multiple times and averaging the results has been done to ensure that the performance metrics are robust and generalizable.

Model	Accuracy		F1 score		Precision		Recall	
	small	large	small	large	small	large	small	large
BERT	0.82	0.95	0.03	0.42	0.01	0.27	0.50	0.97
RoBERTa	0.96	0.93	0.18	0.35	0.12	0.22	0.71	0.97
DeBERTaV3	0.97	0.94	0.26	0.45	0.17	0.30	0.80	0.98
Gemma	0.97	0.97[1]	0.11	0.11[1]	0.06	0.06[1]	0.40	0.40[1]
Mistral	0.91	0.91[1]	0.12	0.12[1]	0.06	0.06[1]	0.80	0.80[1]

Table 4.2: Comparison of model performance metrics.

4.2.1. Small Chromium

Given the metrics defined in the above section and the transformer models in scope, the following observations can be made: BERT appears to be the least effective model for the classification task, with the lowest f1 score, precision, and a moderate recall. RoBERTa's f1 score, precision, and recall are significantly higher than BERT, with improvements being visible both in the identification of security and non-security issues. However, precision is still relatively low, at only 0.12. DeBERTaV3 appears to be the most balanced model with improved marks across all four metrics. DeBERTaV3 achieves a 0.80 recall and 0.17 precision. The recall shows that it can identify 8 out of 10 security issues correctly (true positives), with the rest being false negatives. However, the relatively low precision of 0.17 means that for each correctly labeled security issue (true positive) there are roughly 5 normal bugs classified as security (false positives). For small prediction tasks, this would not be a big impediment. However, if one is to use the model on a larger database, the additional effort required from a validator to cover these false positives would make the model difficult to implement.

[1] Metrics are not impacted as general purpose models are using few-shot prompting that does not differ between datasets

Insight 1: The DeBERTaV3 model is the best-performing model on the small Chromium dataset. Although the recall metric is promising at 0.80, the low precision of 0.17 hinders the effective applicability of the model. The metrics point to a need to improve the ability of the model to detect non-security issues, which could be resolved with a larger, higher-quality dataset.

Looking at the general purpose models, Gemma, despite its high accuracy of 0.97, has a low f1 score at 0.11 defined by a precision of 0.06 and a recall of 0.40, suggesting it may not be as reliable when it comes to the balanced performance necessary for security issue detection. Mistral has the same precision as Gemma at 0.06 but the recall has increased to 0.80.

Compared to the fine-tuned models, some limitations of the general purpose models have to be noted. First of all, through few-shot-prompting it is difficult to control the prediction format. For example, although in the few-shot prompt (see Section 3.2.1) we have indicated that desired results are ”security” or ”non-security”, the model sometimes gives the output as a different sequence, making it difficult to interpret the results automatically. These outputs that are astray from the few-shot prompt examples can be called hallucinations. Important to note that Mistral produced hallucinations in 3% of its outputs and failed to correctly classify bugs as security or non-security in the allowed tokens output. Gemma did not produce hallucinations but was not always able to follow the desired output in the few-shot prompt. From a practical standpoint, this would mean that besides the false positives that the model gives, a security researcher would also have to filter through an additional 3% of the dataset and to classify the issues. Considering these issues, while Mistral’s other metrics have improved compared to Gemma’s, the presence of hallucinations and classification failures could substantially undermine its utility in practice.

Furthermore, general purpose models were also slower at the classification task. The slower classification speed of these models coupled with the necessity for post-processing raises concerns about the scalability of deploying such few-shot-prompted general purpose models in diverse contexts, especially where rapid and accurate security issue detection is paramount. Therefore, while LLMs like Mistral show promise in certain performance aspects, especially due to them not requiring fine-tuning, their practical application on a larger scale warrants a thorough consideration of the trade-offs between speed, accuracy, and the resources required for post-processing.

Insight 2: The performance of the general purpose models using few-shot-prompting is lower than that of the fine-tuned BERT models. Among the evaluated, a notable difference is observed in the recall metric, with Mistral achieving superior recall (0.80) over Gemma (0.40). This suggests that Mistral is better at identifying a higher proportion of actual security issues. However, this comes without an improvement in precision, which remains low at 0.06, the same as Gemma. Consequently, the general purpose models are likely not an effective method for the classification of security issues, at least not without additional fine-tuning.

In conclusion, after the analysis of the performance metrics, DeBERTaV3 emerges as the most effective model for the issue classification task on the small Chromium dataset. Its robust recall paves a promising avenue for identifying security issues, however, the low precision raises questions on the effective applicability of the model in real-life scenarios, where the time of validation from security experts is limited. Compared to general purpose models, its speed in classifying issues is a significant advantage, enhancing its practicality in real-world applications where time is often a critical factor. The lower incidence of false positives compared to general purpose models reduces the operational burden and decreases the costs associated with addressing incorrect false-positive labels. These cost savings are particularly valuable in large-scale deployments where the volume of data to be processed can be substantial. Hence, from the analyzed models, DeBERTaV3 represents the optimal choice. However, all transformer models struggled with achieving a high precision metric. As described by Wu et al. [21], having high recall and low precision means the models are good at finding all the relevant security issues of the dataset, but are not discriminating well between the security and non-security instances. The 0.17 precision of DeBERTaV3 cannot be considered satisfactory and should be higher in order to consider the model adequate for real-life situation deployment.

In the following section, we will investigate the feasibility of using the transformer models on the large Chromium dataset, and observe the changes in performance metrics.

4.2.2. Large Chromium

The large Chromium dataset, which contains a greater number of entries, has enhanced the potential for more effective model training. Observations from the large Chromium dataset include:

The BERT model demonstrated a significant performance increase, with precision rising from 0.01 to 0.27 and recall escalating from 0.50 to 0.97. For RoBERTa, there was also an increase in precision to 0.22, though it is lower than BERT's, while recall improved to the same level of 0.97. This marks a shift in the performance dynamics between the models, where BERT now outperforms RoBERTa. Although RoBERTa exhibited superior performance on a smaller dataset, its effectiveness diminishes in comparison to BERT when applied to larger datasets. This shift could be indicative of each model's capability to handle the complexity and variety inherent in larger datasets. BERT's higher accuracy and f1 score on the large dataset suggest that it may be more resilient to scaling, thus offering more consistent performance regardless of dataset size.

The DeBERTaV3 model continues to lead in performance metrics, achieving the highest f1 score of 0.26 and the highest precision at 0.30, along with an impressive recall of 0.98. Its accuracy, only slightly below BERT's 0.95, remains robust. Moving forward, DeBERTaV3 exhibits promising results for predicting security issues, especially due to the increase in recall from 0.80 on the smaller dataset to 0.98 on the larger dataset. This is critical as it enhances confidence in the model's ability to detect relevant security issues. The precision also improved significantly, from 0.17 to 0.30, indicating a reduction in false positives.

Insight 3: The fine-tuning of the models on the large Chromium dataset with more security entries significantly increased the performance of all models. The most notable improvement was observed in the BERT model, where precision increased from 0.01 to 0.27, and recall rose from 0.5 to 0.98.

Insight 4: DeBERTaV3 continues to outperform all other models, with precision rising from 0.17 to 0.30 and recall from 0.80 to 0.98. The model has enhanced its ability to identify non-security issues. However, the precision level may still be considered insufficient for deployment in real-life scenarios, necessitating a strategy to balance the high recall with improved precision.

In conclusion, it is evident that the increase in dataset size and quality has given the transformer models a larger context window and allowed them to understand better how a security and non-security issue is characterized. This has led to a significant increase in all model's performance, including the DeBERTaV3 model of which precision increased from 0.17 to 0.30, and recall increased from 0.80 to 0.98. Although the precision has increased, it is still relatively low. As recall is very high, one can try to change the precision-recall tradeoff, to increase the precision and lower the effort required for the expert validation. In the following section, a method to change the tradeoff will be explored, which will ideally allow for the increase of the precision metric.

4.2.3. Threshold Investigation

Threshold	Accuracy	Precision	Recall	F1 Score
0.50 (base)	0.9795	0.4778	0.9700	0.6403
0.70	0.9842	0.5460	0.9500	0.6934
0.80	0.9864	0.5864	0.9500	0.7252
0.90	0.9894	0.6528	0.9400	0.7705
0.95	0.9928	0.7460	0.9400	0.8319
0.98	0.9962	0.9082	0.8900	0.8990

Table 4.3: Model performance metrics at various thresholds.

A key challenge in optimizing model performance, particularly when it comes to detecting security issues, revolves around the balance between recall and precision. High recall guarantees that the model captures most security issues (true positives), but this may lead to the inclusion of numerous non-security issues (false positives), which reduces precision. On the other hand, high precision ensures that the model is highly accurate in identifying actual security issues, but this accuracy can come at the cost of potentially introducing false negatives and lowering recall.

To navigate this trade-off and particularly to address the need for higher precision as introduced at the

end of section Section 4.2.2, which is critical in reducing the burden of manual review and mitigating the risks of overlooking genuine threats, we will investigate the effects of increasing the threshold value. The threshold of a model determines the probability above which a predicted outcome is classified as a positive instance. By increasing this threshold, we expect that the model will become more conservative in its predictions, potentially increasing precision at the expense of recall.

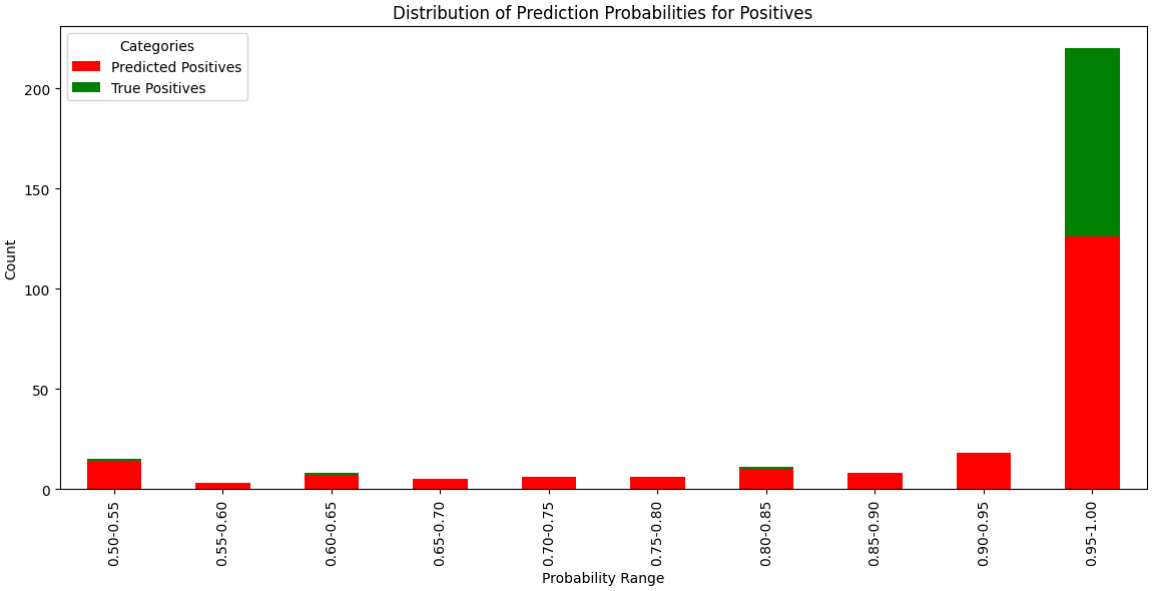

Figure 4.1: Probability distribution of predicted and true positive issues on the large Chromium test dataset

After running the DeBERTaV3 model on the large Chromium dataset a number of 10 times, the model with the best performance characterized by the base entry in Table 4.3 has been picked for evaluation. From the table provided, it is observed that as the threshold increases from the base value of 0.50 to 0.98, there is a consistent improvement in precision. This increase goes from a precision of 0.48 at the base threshold to a striking 0.91 at a threshold of 0.98. However, this increment is accompanied by a decrease in recall, dropping from 0.97 at the base to 0.89. While the drop in recall is notable, the increase in precision is quite significant and may be justified depending on the specific requirements of the task of identifying security issues.

A closer look at the f1 score, which provides a balance between precision and recall, shows an improvement as the threshold is adjusted upwards. The f1 score peaks at a threshold of 0.98, suggesting an optimal balance at this point under the current model's configuration and the dataset it was trained on.

To investigate the tradeoff further, we have plotted the probability distribution of issues over the base threshold of 0.50, to understand where the threshold can be moved in order to avoid missing true security issues. The bars in the figure are stacked, this means that the difference between the height of the red bar and the height of the green bar would be the false positives. A difference of 0 would mean that all of the predicted security issues are true positives. As can be seen in Figure 4.1, most true positives are after the 0.95 probability threshold, with only about 20 issues being false positives, represented by the difference between the red and green bars. Below the threshold of 0.95, only three true positives are present, but a lot of false positives are distributed. This means that by increasing the threshold to 0.95 we can reduce the workload of validating a lot of false positives, and not miss out on many security issues.

Subquestion 1: *Can transformer models be effectively applied to security issue classification, and how does their performance compare to established classification models documented in the literature?*

Answer: The fine-tuned DeBERTaV3 model can be applied effectively using the developed semi-automated security issue prediction pipeline. Using threshold modification, the DeBERTaV3 transformer model achieves performance higher than previous research methods applied on the Chromium dataset described in Table 2.1, achieving an f1 score of up to 0.9.

Therefore, when predicting security issues in other projects, we will apply a post-prediction adjustment, setting the threshold to 0.95. This adjustment aims to enhance the model's precision and reduce the effort required for validation.

Implications: Transformer models demonstrate a robust capability to effectively predict security issues in software development, particularly for C++ projects. While these predictions are not perfect—entailing some false positives that necessitate validation—the performance of these models is sufficiently strong to affirm their real-world applicability across similar projects. However, extending this approach to other programming languages would require access to appropriately labeled datasets for model fine-tuning. If integrated directly into issue tracking systems, the model could potentially assess and classify the security relevance of issues from the initial post in a thread. This capability would enable immediate notifications to relevant stakeholders, facilitating swift validation and prioritization of potential security vulnerabilities. Such an integration could significantly accelerate the process of patching publicly disclosed vulnerabilities, thereby diminishing the window of opportunity for exploitation by malicious actors. Ultimately, this could lead to a more proactive and secure software development lifecycle, enhancing the overall security posture of software systems.

4.3. Predicting Security Labels

4.3.1. Project Selection for Case Study

The selection of a project as a case study for the application of the DeBERTaV3 model is guided by specific criteria to assess its relevance and compatibility with the trained model. Firstly, the project must be developed in the C++ programming language, similar to the language used for the Chromium project on which the model's training was based. This choice is supported by the recommendation of Gegick et al. [47] that the trained model should not be applied to software systems in which the SBRs describe different types of security bugs than those that were used to train the model. Secondly, the project should use security labels for categorizing bug reports to facilitate the comparison of pre and post-validation results. Additionally, the project should have a significant number of assigned CVEs to better understand the difference between coordinated and hidden CVEs. Lastly, the project must be unaffiliated with Chromium to avoid bias from the model training. Table 4.4 presents the top 20 most popular C++ projects by stars (a measure of popularity) on GitHub.

Before choosing a case-study project based on the aforementioned criteria from the table above, we decided to investigate the prediction of security issues and the potential scale of vulnerabilities on four of the top 20 projects with significant CVEs published in the NVD database: Clickhouse, OpenCV, Bitcoin, and gRPC. The DeBERTaV3 model with a 0.95 threshold was applied to predict security issues in the issue trackers of these projects. The number of closed issues analyzed, as depicted in Table 4.5, differs from the total number of closed issues in Table 4.4, as empty issues were removed during preprocessing. As shown in Table 4.5, the ratio of predicted security issues to total issues varies across projects, with ClickHouse having the lowest ratio at 0.8% and gRPC the highest at 2.3%. These findings are intriguing and raise questions about the underlying reasons for this variation, which could be due to the inherent use-case and architecture of the projects or other factors related to the community maintaining the project.

To determine how many of the detected security issues might be unpublished vulnerabilities, expert

[1] Projects do not have any attributed CVEs on the NVD database

Project	Stars	Closed issues	Security issues	Assigned CVEs on NVD
tensorflow	182494	37216	No	428
react-native	115913	24964	Yes	1
electron	111975	18707	Yes	20
terminal	93506	10803	No	7
godot	83640	39475	No	2
bitcoin	76021	7571	No	3
opencv	75616	8029	No	34
swift	65931	8416	No	20
gpt4all	64573	1109	No	0[1]
protobuf	63662	5644	No	4
tesseract	58083	2210	No	2
llama.cpp	56939	2458	No	0[1]
imgui	55785	4743	No	0[1]
Magisk	44197	5541	No	0[1]
x64dbg	43201	1853	No	0[1]
gRPC	40744	10645	Yes	11
json	40266	2122	Yes	2
CPlusPlusThings	37252	215	No	0[1]
leveldb	35074	530	No	0[1]
ClickHouse	34191	16256	No	20

Table 4.4: Top 20 GitHub C++ project statistics [80]. *Projects do not have any CVEs on the NVD database

Project	Issues analyzed	Predicted security issues	Predicted security issue ratio %
ClickHouse	16013	127	0.8
opencv	7978	106	1.2
bitcoin	7284	175	2.4
gRPC	9906	232	2.3

Table 4.5: Security issue prediction statistics for 4 C++ projects hosted on GitHub

validation is required. Given the time needed for this validation, we chose to validate the predictions of one project. For future studies, it might be beneficial to select a random sample of the four projects and validate the results to better understand the cross-project applicability of the model. The project was selected using the criteria introduced at the beginning of this section. Referring back to the initial criteria, specifically the use of security issues, we excluded 16 of the 20 projects, leaving react-native, electron, gRPC, and json. Electron was excluded because it uses Chromium's rendering engine to enable developers to build cross-platform desktop applications using web technologies. Json was excluded due to its relatively low number of closed issues and only 2 attributed CVEs as published on the NVD website. The remaining candidates were gRPC and react-native. Although react-native has more closed issues, it has only one assigned CVE on NVD and not many security-labeled issues. For this reason, react-native was also excluded. This leaves gRPC as the project chosen for the case study.

During the selection process, an intriguing observation emerged: the utilization of security labels is not a universal practice among projects, even among prominent C++ projects hosted on GitHub. This limitation led to the exclusion of several potential candidates.

gRPC is a modern, open-source, high-performance Remote Procedure Call (RPC) framework that can run in any environment. It can connect services in and across data centers with pluggable support for load balancing, tracing, health checking, and authentication. It is also applicable in the last mile of distributed computing to connect devices, mobile applications, and browsers to backend services. Originally developed by Google, gRPC was made open source in 2015 and is now hosted on GitHub. It evolved from the experience of managing large-scale microservices at Google, where it has been used extensively to connect multiple systems across its vast infrastructure. The project's primary goal is to enable efficient and easy-to-manage service communication in a polyglot environment, where different

services could be written in various programming languages but still need to interact seamlessly. At its core, gRPC uses Protocol Buffers, also known as protobufs, which are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. This choice is pivotal because protobufs ensure that the structure of data is preserved and is small in size, which is crucial for high-performance networks. gRPC is primarily written in C++, with core API implementations in other languages like Java, Python, Go, and more. The project has been chosen for the case study due to the similarity in coding language to the trained model, as well as the popularity of the project and the availability of an extensive bug tracker. The use of C++ for its core can lead to concerns particularly related to memory management, buffer overflows, and other low-level issues that are less prevalent in memory-safe languages like Java or Python. However, C++ allows for high performance and fine control over system resources, crucial for a framework like gRPC that emphasizes speed and efficiency.

The gRPC project, like any large-scale and widely used software, has its share of security issues, which can be attributed to several factors inherent in any technology framework that achieves widespread adoption. Additionally, the very features that make gRPC powerful can also introduce complexities that, if not properly managed, may lead to security vulnerabilities.

HTTP/2 Complexities: gRPC is built on top of HTTP/2, a protocol significantly more complex than its predecessor, HTTP/1.1. HTTP/2 introduces sophisticated features like streams, multiplexing, and server push, which, while improving efficiency and speed, also increase the surface area for potential security vulnerabilities. For example, improper implementation of multiplexing can lead to resource exhaustion attacks.

Serialization and Deserialization: gRPC uses Protocol Buffers (protobuf) for data serialization and deserialization. While protobuf is efficient and effective for structured data, serialization and deserialization are common areas where applications can be vulnerable to issues like buffer overflow, especially if not handled correctly.

Extensibility and Interceptors: gRPC's extensibility through interceptors (which can modify request and response behavior) is a powerful feature, but it also introduces risks if interceptors are misconfigured or maliciously implemented, potentially exposing sensitive data or inadvertently bypassing security checks.

4.3.2. gRPC Preliminary Analysis and Prediction

The dataset focusing on gRPC was retrieved on March 14, 2024, utilizing the GitHub API as detailed in the methodology section. It comprises of closed issues from the GitHub issue tracker of the project, deliberately excluding pull requests to concentrate exclusively on issue posts. Each issue entry in the dataset includes only the initial post, deliberately excluding any subsequent discussions. This approach centers on the classification of the initial post with the aim of assessing whether the DeBERTaV3 model can promptly identify a security issue upon the creation of the post. This timely identification is intended to inform relevant stakeholders effectively.

In Table 4.6 the known security aspects of the dataset are detailed. From a total of 9906 issues in the dataset extracted from the issue tracker, 171 have been specifically labeled as security-related by the project's maintainers. Using a regex search across all issues, a total of 11 issues mentioned a published CVE vulnerability, out of which 10 had a direct impact on the gRPC project or any of the downstream applications that it uses. Surprisingly, upon manual verification, it was determined that none of the 10 issues were marked with the security label by maintainers.

Total Issues	Security labeled	Total CVE mentions	Total CVE relevant
9906	171	11	10

Table 4.6: gRPC dataset initial analysis statistics

For predicting security issues within the gRPC dataset, the issues were preprocessed according to the methodology outlined in Section 3.2.2. Using a DeBERTaV3 model, which is characterized by the base performance outlined in Table 4.3, predictions for security issues were made for the gRPC dataset. Modifications to the model's threshold settings lead to changes in the number of predicted issues. As shown in Table 4.7, the number of issues predicted as security-related varies from 573 to

183 across different thresholds, reflecting the trade-off between recall and precision. This variability underscores the potential burden placed on security experts to validate predicted issues, particularly in large projects like gRPC, where validating a significant volume of predicted security issues requires considerable effort. Due to the demonstrated effectiveness of a 0.95 threshold of the DeBERTaV3 model in managing the precision-recall trade-off in section Section 4.2.3, the dataset of 232 issues was selected for delivery to experts for vulnerability validation.

Threshold	Predicted security issues
0.50 (base)	573
0.95[1]	232
0.98	183

Table 4.7: gRPC predicted security issues based on threshold value

4.4. Expert Validation

For the validation of the findings, a security expert who also holds the role of a CNA was involved. The expert was provided with a dataset containing the links to 232 predicted security issues within the gRPC project. These security-related issues were assumed to be easier for the model to predict due to the availability of the dataset for training. However, it is important to note that while CVEs are a subset of these security-related issues, the model was not specifically trained to predict CVEs due to the limited dataset size and the difficulty of predicting CVEs from just bug descriptions.

The expert evaluated each security issue based on the entire discussion and determined, using their expert knowledge, whether the issue would qualify as a CVE vulnerability. This process helped filter out non-security issues (false positives) and security issues that were feature requests or questions. Additionally, the expert provided an estimation of the CVSS score based on the CVSS 3.1 framework described in the methodology section. Lastly, the expert offered additional comments to highlight any other important aspects, such as duplication of issues or reasons for non-qualification. Table 4.8 presents a summary of the validation results.

Predicted security issues	CVEs	Mean CVSS 3.1 score
232	52	5.3

Table 4.8: Validation results of the predicted gRPC issues using the 0.95 threshold

The validation results are noteworthy. Out of the 232 security issues flagged by the model, 52 have been validated as potential CVEs. Interestingly, the National Vulnerability Database lists only 11 CVEs under the gRPC project. This discrepancy underscores the efficacy of the model in uncovering hidden CVEs, with the model identifying nearly five times more CVEs than what is currently listed. It should be noted, however, that this analysis only covers issues flagged by the model, implying that the actual number of hidden security issues could be even higher. This hypothesis is supported by the validation process, where the expert identified that three CVEs were duplicates of similar issues missed by the model. For example, *issue 11238*, marked as a CVE with a CVSS 3.1 score of 6.5, is a duplicate of *issue 11015*, which was not flagged as a security issue by the model.

Subquestion 2: *How prevalent are untracked vulnerabilities in widely-used open-source C++ projects on GitHub, and what are their common characteristics?*

Answer: The validation of the predictions on the gRPC dataset revealed 52 potential hidden vulnerabilities. Considering the size of the initial dataset of 9906 entries, a potential vulnerability is published in an issue tracker every 190 issues. Because the issue trackers of projects varies in size, an estimation based on the results of the gRPC validation is that approximately 0.5% of the published issues in the public tracker of a C++ project on GitHub are potential vulnerabilities. For C++ projects, the most common untracked vulnerabilities are related to memory issues.

[1] Threshold value used in the next section to adjust the prediction results of DeBERTaV3 model

Below is an example of an issue that has been classified as security by the model and validated as a CVE by the expert, *Issue 21927* with a CVSS 3.1 score of 7.1. The issue reported in the gRPC repository involved a potential out-of-bounds memory access in the *trim()* function defined in *check_gcp_environment.cc*. This function did not properly handle cases where the input string was empty, causing an underflow in a variable that could lead to unauthorized memory access if certain conditions were met. This is a type of buffer overflow vulnerability, which can potentially allow an attacker to read or corrupt memory outside of the buffer's bounds, leading to crashes, data corruption, or the execution of malicious code. The issue was classified as a bug with moderate priority and was fixed by adding a boundary check to handle the empty string case properly, mitigating the risk of exploiting this vulnerability.

Issue 21927: "The trim() function defined in the following file doesn't correctly handle the edge case of an empty string. https://github.com/gRPC/gRPC/blob/master/src/core/lib/security/credentials/alts/check_gcp_environment.cc The end variable underflows to the max long value when strlen(src) -1 is evaluated; and memory outside allocated buffer is accessed to be checked for a space. If couple of spaces exist at that point in memory, the returned string would leak contents of the memory. To fix, add a check for edgecase where strlen(src) == 0 and return src as is in such a scenario."

> **Implications:** Untracked vulnerabilities identified in issue trackers represent a significant security risk. With a security vulnerability emerging every couple of hundred posted issues, the frequency of such exposures can be alarmingly high, especially in active project communities. This rate potentially translates to at least one security vulnerability being disclosed each month within the issue tracker. Critically, these vulnerabilities often remain untracked, meaning that even after they are addressed within the original project, downstream projects that integrate this software may not be notified about the necessary security patches. As a result, they might continue operating with vulnerable software unknowingly. This situation is further amplified by hackers who, knowing which versions of software remain unpatched, can conduct targeted reconnaissance to exploit these weaknesses. Consequently, these vulnerabilities can propagate across the entire software supply chain, challenging coordination efforts due to the absence of a centralized tracking mechanism like CVE. The lack of such a mechanism not only impedes effective vulnerability management but also hinders the timely dissemination of patch information, thereby amplifying the risks to all software users and stakeholders involved.

It is also valuable to explore whether any of the detected vulnerabilities correspond to existing CVEs in the NVD database for the gRPC project. A manual analysis of the 11 posted CVEs on NVD was conducted to identify any references back to the GitHub repository. The analysis revealed that few issues were mentioned in the CVE descriptions, which typically reference pull requests rather than direct issues. None of these referenced issues matched those validated as CVEs by the expert, indicating no duplication between the currently published CVEs and those identified by the model. This brings the total number of identified CVEs for the gRPC project to 63 (coordinated plus non-coordinated).

Another point of interest is the correlation between identified CVEs and the use of security labels in the repository. The gRPC project employs a label called "area/security." An investigation into the identified CVEs and their label tags revealed that only 3 out of the 52 identified CVEs used the security label. This discrepancy highlights a gap between the type of issues flagged and the labeling process in the repository.

With a comprehensive view of the scale of CVEs in the gRPC project and the contrast between developer perceptions (as indicated by security labels) and the actual situation, it is essential to discuss the severity of the identified CVEs. The most common CVSS score in the dataset is 5.1, appearing 44 times, as can be seen in Figure 4.2. This suggests that many of the vulnerabilities are of moderate severity, which, while not immediately critical, still require attention to prevent potential exploitation. The consistency in these scores indicates a uniformity in the types of vulnerabilities being reported, possibly within a similar context.

However, several outliers deviate from the predominant score of 5.1, including scores of 5.9, 6.3, 6.5, and 7.1. These higher scores represent more severe vulnerabilities that could have significant impacts if exploited. For instance, a CVSS score of 7.1 denotes a high severity, suggesting a vulnerability that could be more easily exploited or lead to more damaging consequences. The presence of these

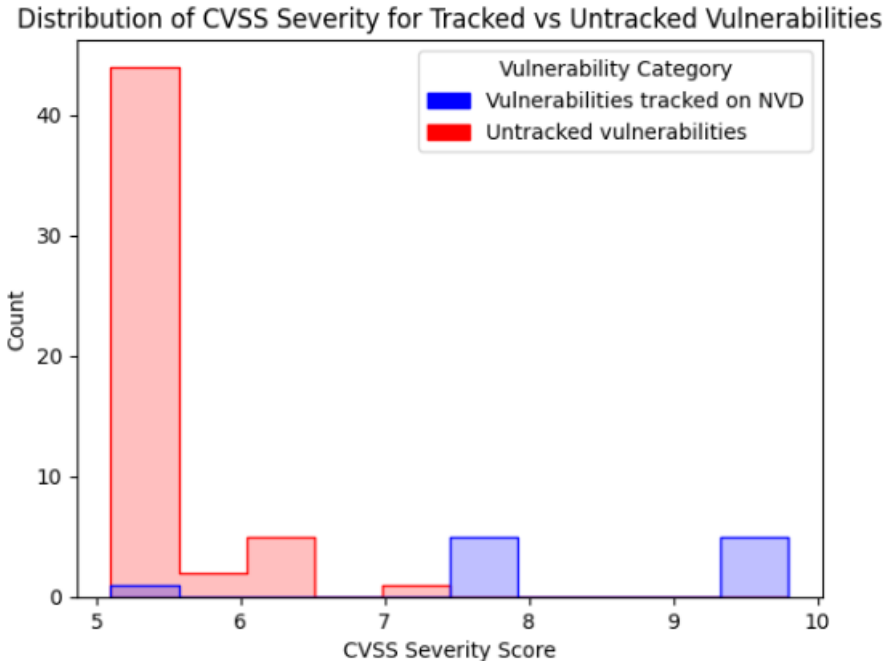

Figure 4.2: CVSS rating distribution of tracked/untracked vulnerabilities for the gRPC project

outliers underscores the necessity for a nuanced approach to vulnerability management, prioritizing higher-risk issues for remediation. The average CVSS 3.1 score of the issues is 5.3, due to the high number of issues scoring 5.1. Compared to the published CVEs on NVD, the scores of the identified vulnerabilities is lower.

There is a notable disparity in the CVSS scores between tracked and untracked vulnerabilities, with those cataloged in the National Vulnerability Database generally receiving higher ratings, as can be seen in Figure 4.2. This discrepancy may stem from several factors. Primarily, the detailed reporting process facilitated by security researchers and the additions of the NVD likely results in more comprehensive assessments of the vulnerabilities' metrics. Conversely, initial issue posts might not fully capture the complexity of the vulnerability taxonomy, often leading to lower assessed CVSS scores. Additionally, security researchers may have incentives to secure higher severity ratings for their findings to underscore their impact and urgency, contrasting with untracked vulnerabilities, which are typically disclosed merely to expedite the development of a security patch.

Among the 52 CVEs, three have been identified as having known exploits: issues *7227*, *22506*, and *35383*. The identification of these CVEs with exploits is critical as it provides actionable intelligence for security teams. Knowing which unpatched vulnerabilities are published in the wild with exploits allows organizations to prioritize patching and mitigation efforts more effectively. The range of severity scores among these exploited CVEs highlights that even moderately rated vulnerabilities can be targeted by attackers. This is particularly dangerous because malicious actors can exploit these vulnerabilities to gain unauthorized access, disrupt services, or steal sensitive information, posing significant risks to organizations.

Subquestion 3: *What is the potential impact of these hidden vulnerabilities on the security of open-source software?*

Answer: The research has uncovered that the average CVSS 3.1 score of the vulnerabilities identified in the gRPC project is 5.3, which suggests a moderate level of threat. The potential impact of untracked vulnerabilities is lower than that of tracked vulnerabilities on NVD. This is a good sign, showing that critical vulnerabilities are disclosed privately and patched before public disclosure. However, the average CVSS score of untracked vulnerabilities may obscure the potential severity of certain vulnerabilities that, if exploited, could compromise the software's integrity and data security. The first issue post of a vulnerability may not be providing all the details of the taxonomy to make the vulnerability critical, leading to it receiving a lower CVSS score when validated. The lack of consistent labeling and documentation discrepancies between identified issues and those recorded in official databases like the NVD highlights a critical need for improved tracking and reporting systems in open-source projects.

As we have explored the issues validated as CVEs, it is equally important to examine the non-CVE issues identified by the security expert. The expert's additional comments reveal that several non-CVE issues pertain to memory leaks. Interestingly, these occur at process shutdown and lack security implications. Additionally, some issues were misclassified as security-related, likely due to the presence of keywords typically associated with security concerns, such as 'exploit,' 'PoC,' or 'vulnerability.' Moreover, certain issues, while related to security, did not impact overall security; these were primarily feature requests or transient bugs. These findings underscore the critical role of expert validation in sifting through potential false positives and refining the accuracy of issue classification.

4.5. Cross-language Application

In the previous section, we discussed the results of the Chromium model, fine-tuned for predicting issues in the gRPC dataset, with both projects being developed in the same programming language, C++. In this section, we will briefly explore the performance of this C++ fine-tuned model when applied to datasets of projects written in two other programming languages, Java and Python. This exploration involves testing the model on four datasets curated by Wu et al. [21]. The hypothesis is that the Chromium fine-tuned model may underperform on these datasets due to differences in programming languages and their associated security vulnerabilities. The datasets, which are publicly available [81] and presented in Table 4.9, have been pre-processed as described in Section 3.2.2. Predictions were made using the DeBERTaV3 model, characterized by the base performance metrics shown in Table 4.3.

Project	Languages	SBR	NSBR	Accuracy	Precision	Recall	F1 Score
Ambari	Java + Python	56	944	0.92	0.28	0.23	0.26
Camel	Java	74	926	0.91	0.29	0.19	0.23
Derby	Java	179	821	0.82	0.52	0.21	0.30
Wicket	Java	47	953	0.93	0.16	0.15	0.15

Table 4.9: Combined dataset and performance metrics for cross-language projects

Based on the results presented in Table 4.9, it is evident that the DeBERTaV3 model, fine-tuned on the Chromium dataset, performs poorly when applied to projects written in different programming languages. Across all evaluated projects, recall remains low, with the highest value of 0.23 recorded on the Ambari project. Such a low recall indicates that the majority of security issues could be missed by the model. Additionally, the precision ranges from 0.16 for Wicket to 0.52 for Derby, indicating a significant number of false positives. This level of precision could render the model impractical for real-world application due to the high rate of incorrect security issue predictions. Although this cross-language evaluation predominantly included Java-based projects, the results suggest that similar outcomes would likely occur with other programming languages. Unfortunately, the absence of publicly available, high-quality, curated datasets of vulnerability reports makes it impossible to confirm this with certainty. These limitations of the model are further discussed in Section 5.2. This situation underscores the necessity for

developing models that are fine-tuned on datasets closely matching the programming language of the target project to achieve optimal results.

> **Implications:** The fine-tuned model is currently optimized for application within C++ projects. Security issues inherent to projects developed in other programming languages often differ significantly from those in C++, which can lead to suboptimal prediction performance when the model is applied outside its training context. To ensure robust performance, it is crucial that the model be fine-tuned on datasets specifically curated for the target programming language. However, the availability of high-quality, labeled datasets for software vulnerability descriptions is currently limited, posing a significant challenge to this approach. There is a pressing need for the development and curation of comprehensive datasets encompassing a variety of programming languages to enhance the adaptability and effectiveness of predictive models in identifying software vulnerabilities.

4.6. Results Discussion

Investigating non-coordinated vulnerabilities in open-source software is critical for maintaining digital security. Open-source software, known for its adaptability and strong community support, also introduces potential risks. The open nature of such software allows for rapid updates and collaborative problem-solving, yet it also exposes vulnerabilities directly on public platforms like GitHub, accessible to everyone, including potential attackers.

The ramifications of these vulnerabilities, if exploited, can be significant. They might lead to the theft of sensitive data, disruptions in service, and substantial financial and reputational damages to organizations that depend on these open-source projects. Historical incidents have vividly illustrated the tangible impacts of such security breaches, highlighting the urgent need for focused research in this area.

Moreover, the inherent transparency of open-source software, while a benefit for collaborative development, also serves as a catalyst for cyber threats. Every publicly disclosed vulnerability coupled with a missing patch can act as a roadmap for attackers, guiding them to exploit weaknesses before they are fully addressed.

Additionally, the exploitation of such vulnerabilities can escalate beyond direct impacts on the originating organization, affecting entire supply chains and consumer bases. The interconnected nature of digital services means that a breach in one open-source component can ripple across multiple systems, amplifying the damage and complicating mitigation efforts. Past security incidents linked to open-source vulnerabilities, such as the infamous Heartbleed bug [82], underscore the extensive consequences that can arise from such exploits and reinforce the necessity of vigilant security practices within the open-source community.

In the quest to understand the landscape of untracked security vulnerabilities in open-source software hosted on GitHub, this study has illuminated several critical dimensions of cybersecurity practices. The study's application of transformer-based models to identify security-related threads within GitHub issue trackers has demonstrated notable success, revealing vulnerabilities that would otherwise remain undetected by traditional CVE processes. For this task, BERT-based models and the general purpose models of Mistral and Gemma have been put up to the task of security issue classification. The BERT-based models have been fine-tuned on two versions of the Chromium dataset introduced by Wu et al. [21], while Gemma and Mistral have been applied using few-shot prompting. The first version of the Chromium dataset contained 192 SBR, while the second one contained 808 SBR. The difference in the number of SBRs comes from security experts who manually checked the issue tracker and changed the labels of issues such as memory leakage and null pointer problems which were initially marked as NSBR to SBR. This choice has been taken as the aforementioned issue types belong to classic vulnerability types since such issues are exploited by hackers [21] and have been listed in the top 25 most dangerous Common Weakness Enumeration (CWE) types [83]. It is important to mention that by using the dataset of Wu et al. [21], we are training a model to predict security-related issues and not CVE vulnerabilities. A SBR does not have to be a CVE necessarily, it can also be a feature request related to security, or a bug, without an attached exploit. This design choice means that after the prediction of SBR, the results had to be validated by an expert to identify which of the SBRs are CVEs. This adds an

extra layer of effort to the vulnerability identification, however, it appears to be the most effective way forward to deal with issues being flagged false positives. Directly identifying CVEs using transformers would require a database of issues that qualify as CVEs and is large enough to avoid problems such as data imbalance. Although one could construct a database using CVE descriptions from the NVD database, the description has been altered and processed and does not reflect what a first post issue on an issue tracker resembles. Back-tracking published CVEs to the initial post on GitHub would be a way to construct a CVE database that could reduce the number of non-CVE SBR predictions.

The fine-tuned DeBERTaV3 model has demonstrated high-performance metrics on the test dataset for the task of detecting security issues and outperformed the BERT and RoBERTa models, achieving an f1 score of up to 0.90, with threshold modification on the large Chromium dataset. The performance of the DeBERTaV3 model is higher than any of the cited methods that have been applied on the dataset curated by Wu et al. [21], including FARSEC [48], Random Forest [21, 49] and Active Learning [56]. These results mean that given a strong enough dataset, transformers represent a state-of-the-art method for security issue classification.

The DeBERTaV3 model has demonstrated good cross-project applicability, identifying 232 security issues from an issue tracker of 9906 issues of the gRPC project. The main findings underscore a significant prevalence of security vulnerabilities that are not captured within the conventional CVE system. From validation with experts, 52 hidden CVE vulnerabilities have been revealed, a significant increase compared to the 11 CVEs published on the NVD database. Furthermore, the CVEs had an average CVSS 3.1 score of 5.3. These results are particularly important in the context of existing literature that often focuses predominantly on coordinated vulnerabilities, thus overlooking the vast array of untracked issues that this research has brought to light. A critical insight from this study is the identification of significant gaps in the current CVE assignment processes. All of the vulnerabilities detected by the model did not correspond to any recorded CVE entries, suggesting either a lag in the CVE assignment or a complete oversight. This finding aligns with concerns raised by prior studies which suggest that the CVE system, while robust, is not exhaustive in capturing all existing vulnerabilities, particularly those found in less prominent or rapidly evolving projects. The difference between published and hidden vulnerabilities not only challenges the existing security frameworks but also calls into question the effectiveness of current vulnerability management strategies deployed across open-source ecosystems. An important limitation to mention is the validation of only SBR predictions by the expert. It remains uncertain how many more vulnerabilities that are genuinely security-related might be incorrectly labeled as non-security-related and overseen in the validation process. The study of Wu et al [21], found that the SBR class for the Chromium dataset increased in size from 192 to 808 through the expert validation of the entire dataset, which points to the possibility of there being mislabeled NSBR issues, and ultimately more hidden CVEs in the gRPC results. The percentage of security issues predicted in the gRPC issue tracker is 2.3%, which is close to the 3% ratio that Zahedi et al. [36] found in their research to be an estimation for GitHub projects. Although the percentage is lower than the author's findings, it still shows that security issues are a restricted subset of the issue tracker. The ratio of 2.3% is when considering the 232 flagged SBRs from the model with a 0.95 threshold. The change in threshold is also directly related to the ratio. The ratio is inversely proportional to the threshold value. With the basic threshold of 0.50, the model predicts 573 security issues, and the ratio would be equal to 5.7%. However, in practice, the ratio would be smaller, given that these issues have also not been validated as security-related by the expert, and are only the predictions of the model.

Building on the significant findings revealed by the DeBERTaV3 model, it is imperative to grasp the broader implications for the vast landscape of C++ projects across GitHub. The discovery that within the gRPC issue tracker of 9906 issues, there is a total of 52 hidden vulnerabilities, this means that for every 190 issues, there is one vulnerability disclosed publicly. This aspect in the gRPC project highlights a potential underreporting and detection gap that might be reflective of a broader trend within the open-source C++ repositories. If this rate is indicative of similar C++ projects on GitHub, the implications are alarming. The analysis of the top 20 most popular C++ projects as introduced in Table 4.4, reflecting a total of 208,607 closed issues, presents a unique perspective on vulnerability management in these high-profile repositories. Considering the frequency of 1 vulnerability every 157 posted issues, this would total approximately 1100 total vulnerabilities, which is considerably higher than the total 554 vulnerabilities currently assigned to these projects on NVD. Furthermore, considering a conservative issue posting rate of 3 issues per day for each of these top 20 projects, approximately 60 issues are

posted per day, which would result in one vulnerability disclosed publicly in an issue tracker every 3 days. This figure underscores a critical gap in the current CVE system's ability to capture and report vulnerabilities within major software projects. The findings highlight the pressing need for more advanced, AI-driven tools, like the DeBERTaV3 model, to be deployed for proactive vulnerability detection. Such tools could significantly improve the detection rates of hidden vulnerabilities, thereby enhancing the security posture of these critical projects that are used in the current software infrastructure.

The discrepancy between coordinated and hidden vulnerabilities in bug repositories is probably due to an accumulation of problems in the vulnerability disclosure cycle [14]. On the one hand, developers (discoverers) are not following best practices and they are posting vulnerability-related issues, including exploits on issue trackers [36], instead of coordinating with the maintainer and CNA. Subsequently, these security vulnerabilities expose the system to attackers. Understanding these publicly posted issues can help in identifying common vulnerabilities and solutions, thereby contributing to more secure software development practices.

On the other hand, the number of CVE IDs has rapidly increased over the years, putting pressure on the resources available to CNAs [84]. The significant increase in the creation of CVE IDs year over year is expected to continue indefinitely. Originally, the CVE program was hampered by limited resources, but recent decoupling efforts have allowed for a substantial increase in CVE submissions by CVE Numbering Authorities. These efforts have enabled a scalable solution to support the program's growth. Despite this scalability, the CVE program still maintains oversight of CNAs to ensure adherence to proper procedures, content quality, and consistency. A new resource constraint has emerged downstream, particularly at the National Vulnerability Database (NVD) level, where analysts manually add extensive metadata to each CVE. This process, which includes assigning scores and tagging vulnerabilities, is resource-intensive and not sustainable with the current staffing, especially as the number of CVEs continues to grow [84]. Although some entities provide basic NVD metadata, there are no standardized policies to ensure consistency across all CVE records. To address these challenges, the CVE program has introduced the Collaborative Vulnerability Metadata Acceptance Process (CVMAP). This process allows external entities to submit CVE record metadata with minimal involvement from NVD analysts, who then act more as auditors to ensure standards of transparency and consistency are maintained. This system incentivizes participation by displaying the contribution level of participants on the public NVD website and allowing CNAs to set initial CVSS vector strings and communicate their findings directly to the user community. This approach aims to utilize the technical expertise of CNAs more effectively and relieve some of the burdens on NVD analysts by fostering a more collaborative and efficient environment for managing CVE metadata. While it's challenging to predict the exact growth of CVEs and the participation levels in this submission process, the initiative is expected to enhance the accuracy of vulnerability metadata and alleviate the growing workload at the NVD [84].

Through the fine-tuning and testing of the BERT-based models on the two Chromium datasets, it can be seen that dataset quality and size are crucial aspects of the performance of the model, especially on more basic models like BERT and RoBERTa. For example, the f1 score of the BERT model increased from 0.03 on the small Chromium dataset to 0.42 on the large Chromium dataset. An increase can be observed also in the RoBERTa model, with almost a doubling in f1 score, from 0.18 to 0.35. These findings point to known challenges in the task of vulnerability prediction. The increase in the f1 score supports the need for data quality, as mislabeling of issues in the training dataset can lead to poor performance in SBR prediction models [21]. Furthermore, the large Chromium dataset also helps to tackle data imbalance which undermines the effectiveness of models by misrepresenting the true likelihood of security issues [62]. Interesting to note is the application of the threshold modification in order to improve the performance metrics of the models. In the results section, we present that most SBRs were predicted with high confidence, with the probability of being SBR predicted in the confidence interval of 0.95 to 1.00. For the SBR instances, the model clearly identifies features or patterns in the text that strongly align with the characteristics of the security issues. This confidence is typically a good sign of model effectiveness, but it could also indicate overfitting, especially if these high confidence scores are not mirrored by similar performance on a validation or test set. However, during model training the performance metrics have been evaluated across epochs and overfitting has been avoided by setting an early stopping callback function. This aspect allows us to say that the model has learned to identify successfully security issues in the Chromium dataset. In the practical application of identifying CVEs from the SBR predictions, it is important to note the limitation in the validator's

available effort. The high confidence in the positive (SBR) class has influenced the decision threshold setting. For example, the high confidence score reliably indicates a correct prediction and allows for the increase in the threshold to reduce the number of false positives, and ultimately the validation time. This has been a conscious decision and refers back to the tradeoff between precision and recall that was introduced in the paper.

The motivation to explore Gemma and Mistral, two of the latest general purpose models, in the task of security issue classification arises from their superior performance on standard benchmarks over BERT-based models [22, 23]. Besides their advancements, these models hold the potential to identify security issues using few-shot prompting without the need for fine-tuning with labeled datasets—a significant advantage given the scarcity and low quality of such datasets [21]. This capability could greatly simplify the classification process if effectively implemented. However, the practical application of Gemma and Mistral using zero-shot and few-shot techniques has encountered notable challenges. Firstly, controlling the output format of these models is problematic. Zero-shot prompts often yield varied results in terms of output length, word sequence, and format consistency. While few-shot prompting has shown some improvement in controlling outputs, it is still insufficient to prevent model hallucinations. These hallucinations include failing to specify whether an issue is security-related, providing excessive explanations without clear conclusions, or contradictorily classifying an issue as both security and non-security-related. Adjustments to parameters like temperature have been made to curb the models' creativity and standardize outputs, yet these measures have largely been unsuccessful. This difficulty in controlling outputs highlights a fundamental limitation in the current design of general purpose models for large-scale security issue identification. The expansive context window of these state-of-the-art models had initially suggested a promising avenue for leveraging their inherent capabilities in security classification. However, given the challenges encountered, alternative approaches may need consideration. One potential route is to fine-tune the base models using methodologies similar to those employed for the DeBERTaV3 model. Such fine-tuning is expected to further enhance the performance of Gemma and Mistral beyond that of the existing BERT-based approaches. Unfortunately, the high computational demands and resource constraints of this master thesis precluded the possibility of fine-tuning, pointing to an area for future research and application.

4.7. Policy Implications

The challenge of managing hidden Common Vulnerabilities and Exposures in open-source software is only one part of what constitutes the grand challenge of cybersecurity. A comprehensive and collaborative strategy involving multiple stakeholders is essential to improve the identification, reporting, and patching of these vulnerabilities.

The CVE system needs a more dynamic and collaborative framework to quickly identify and address vulnerabilities. This involves improving the cooperation between software developers, security experts, project maintainers, and CVE Numbering Authorities. To do this, we will discuss possible interventions to hopefully introduce change in the disclosure process in open-source software taking into account each of the stakeholders introduced in Section 2.6. These changes will focus specifically on the interaction between two influential stakeholders, the discoverers and software vendors (referred to as project maintainers). These two stakeholders are involved in the process of vulnerability reporting, as displayed in Figure 2.2.

One of the biggest aspects of the challenge at hand is the disclosure of CVE vulnerabilities in public trackers by discoverers, and exposing sensitive information to the public. When discussing discoverers, we think of developers and security researchers. It is assumed that security researchers are aware of the coordinated vulnerability disclosure process and follow the ideal information flow introduced by Lin et al. [14]. However, developers are probably the majority posting sensitive CVE information in the public issue tracker. To effectively address the issue of developers posting CVE vulnerabilities in public issue trackers, it is crucial to understand their motivations and the systemic pressures they face. Developers may post vulnerabilities publicly due to a lack of awareness about the security implications, the absence of a clear protocol for secure communication, or out of frustration with slower, bureaucratic processes. The ideal target is to induce behavior change and prevent developers from posting CVE-related issues in the issue tracker. In addressing the challenge of changing the developer's behavior, we propose the use of the COM-B (Capability, Opportunity, Motivation – Behaviour) framework provides

a structured approach to understanding behavior and how to change it [85]. The COM-B system posits that behavior (B) is part of a system involving interactions between one's capabilities (C), opportunities (O), and motivations (M). In the following rows, intervention steps will be explained, and the impact of the changes on the coordinated vulnerability disclosure process will be introduced.

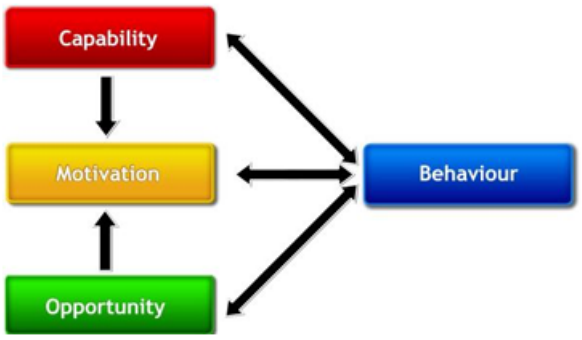

Figure 4.3: COM-B model, adapted from the Behaviour Change Wheel introduced by Mitchie et al. [85]

Opportunity

- **Establish a Vulnerability Disclosure Policy:** The project maintainers should establish clear, accessible guidelines outlining steps for secure vulnerability disclosure, integrated into project onboarding processes. For some collaborative platforms like GitHub, a dedicated Security tab is available, where project owners can post a security disclosure policy, including step-by-step instructions on achieving coordinated vulnerability disclosure. The goal of this intervention step would be to ensure developers understand proper procedures for reporting vulnerabilities to reduce public disclosures due to confusion or lack of information [86].
- **Create a Dedicated Security Channel:** The project maintainers should set up secure, private communication channels like encrypted emails or private issue trackers for reporting vulnerabilities. The goal of this intervention would be to provide a straightforward way for developers to report vulnerabilities securely.

Motivation

- **Implement a Bug Bounty Program:** The project maintainers should introduce or enhance bug bounty programs with competitive rewards for privately disclosed vulnerabilities. The goal of this intervention would be to financially motivate developers to follow the official disclosure process, rewarding their security contributions, which has been proven to increase the security of open-source software through better vulnerability disclosure [87].
- **Recognize and Celebrate Responsible Disclosure:** The community should publicly acknowledge and praise developers who adhere to disclosure guidelines through various public channels. The goal of this intervention would be to foster a culture that values security, encouraging developers by recognizing responsible behaviors.

Capability

- **Conduct Regular Security Awareness Training:** Project maintainers should provide free-to-access training on security best practices and the importance of secure disclosure and socialize this training across communication channels. The goal of the intervention is to raise awareness among developers about security and their critical role in ecosystem safety.
- **Include Security as Part of the Development Lifecycle:** Developers should integrate security reviews and assessments as standard parts of the development process using a DevSecOps approach. The goal of this intervention would be to proactively embed security considerations into development, promoting a security-first mindset, which has been proven to address security issues early in the development lifecycle, and ultimately reduce the appearance of vulnerabilities [88].

Implementing these recommendations should help create a safer cybersecurity environment by improving methods and cooperation used to manage vulnerabilities. This enhances not only individual projects' security postures but also contributes to the broader ecosystem's resilience against cyber threats.

5

Conclusion

This thesis has analyzed the problem of untracked software vulnerabilities in open-source projects in GitHub repositories, a critical yet mostly underexplored domain within the cybersecurity field. The investigation focused on vulnerabilities that are typically absent from official databases by not being attributed CVE numbers, aiming to uncover and assess the extent of these untracked vulnerabilities. Utilizing a semi-automatic pipeline to extract, predict, and validate security issues into hidden CVEs, this study has provided significant insights into the prevalence and severity of security issues in open-source software. Throughout the research, it was revealed that a considerable number of security vulnerabilities remain undetected or unreported in public issue tracking systems, bypassing the coordinated vulnerability disclosure process. For the identification of security issues in bug repositories, transformer models have been employed. The fine-tuned DeBERTaV3 model, used for classifying GitHub issues, demonstrated good performance metrics, substantiating its utility in identifying latent vulnerabilities. Specifically, the application of this model on the gRPC project uncovered a disparity between the number of identified security issues and those recorded in official databases, with hidden CVEs being five times as many compared to the published CVEs in the NVD database, highlighting the underreporting problem. Based on these results, the research question regarding the extent of untracked vulnerabilities in GitHub projects has been successfully addressed in the research, and the answer can be summarized as follows.

> **Research Question:** *What is the extent of vulnerabilities in open-source projects on GitHub that do not have CVE identification numbers?*
>
> **Answer**: Approximately 0.5% of the issues published on a C++ project bug tracker are possible vulnerabilities that have not been assigned CVE numbers. The severity of these CVEs is medium, and the vulnerabilities are mostly related to memory issues.

During the process of fine-tuning a transformer model for the task of classification of security issues, several challenges such as data quality, dataset size and cross-project applicability have been addressed, all pointing to a problem already known in the domain of cybersecurity: the insufficient size and diversity of vulnerability datasets to allow for better performance of automatic vulnerability identification methods.

The implications of these findings are profound, necessitating a reevaluation of current practices in vulnerability reporting and management. By exposing the discrepancy between coordinated and untracked issues, this study has hopefully informed stakeholders about the need to enhance coordinated vulnerability disclosure within open-source communities. The presence of exploitable vulnerabilities in public issue trackers poses a significant threat not only to the security of the software itself but also to the entire supply chain and end-users who integrate such open-source solutions into their architecture. We propose an initial step toward improving the disclosure process, focusing primarily on the interaction between developers and project maintainers. The proposed interventions aim to shift developer behavior from disclosing security issues in public trackers to coordinating with product maintainers through

more secure channels. With these results, this thesis contributes a crucial piece to the larger puzzle of cybersecurity. By shedding light on hidden CVE vulnerabilities, we hope that addressing this issue will enhance the resilience of open-source software against cyber threats and improve supply chain security.

Throughout this thesis, we have systematically explored the less discussed untracked software vulnerabilities and proposed innovative methodologies for their effective management. As the field of cybersecurity progresses, it becomes crucial to translate these theoretical advances into practical applications. This work represents a single drop in the vast ocean of efforts required to tackle the grand challenge of cybersecurity.

5.1. Scientific and Societal Relevance

In the scientific domain, the proposed research adds to the evolving understanding of open-source software security dynamics, providing empirical evidence and insights for future research in the cybersecurity field. On the topic of security issue identification from bug reports, several authors have made advancements [52, 21, 48]. Although their results are promising, one of the downsides is the lack of performance when applied to other projects, due to the lack of context introduced by the reduced dataset size. The research presented used transformers that have a larger contextual window, with the hypothesis that this will allow for better cross-project applicability of the models. This hypothesis has been confirmed, with the fine-tuned transformer model achieving better performance metrics than previous research on the Chromium dataset. In addition to this, the application of the model to predict security issues on a different project is a new approach in the literature. The application of the model on gRPC demonstrated that security issues can be found effectively from issue trackers using transformer models, effectively filtering through large datasets in order to find and validate potential impactful vulnerabilities.

The societal relevance of the proposed research is significant. It addresses the critical need for robust cybersecurity in an increasingly digital world. The insights from this paper have direct implications for the common person by enhancing the security and reliability of the digital technologies they use. The identified untracked vulnerabilities are lying dormant in the bug tracker of the open-source projects. Without being introduced in a central tracking system such as the CVE system, the vulnerabilities are not being coordinated throughout the supply chain. Even though they might be patched, the downstream applications are not made aware of the possibly insecure version of their software. This means that end-users can end up operating software in their day-to-day life that could be exploited to affect the confidentiality, integrity or availability of their information. The insights from investigating security issues can lead to improved coordination in fixing the vulnerabilities. Ultimately, this will result in fewer security breaches, reduced risk of personal data exposure, and increased trust in digital services [30]. For the average person, this translates into a safer online environment, where the risk of identity theft, financial fraud, and privacy violations is significantly reduced, thereby contributing to a more secure digital life.

5.2. Limitations

The research is subject to several constraints. One of the main limitations is the sole focus on the C++ programming language. The DeBERTaV3 transformer is specifically tuned using the Chromium dataset which utilizes C++. Similarly, the gRPC project, also developed in C++, has been analyzed for the prediction of security issues. This consistent use of C++ may enhance the model's ability to identify security issues characteristic of this language, such as memory-related issues. However, it remains uncertain whether the model would exhibit comparable efficacy on projects coded in other languages like Java or Python. Section 3.3 aimed to showcase that the cross-programming language applicability of the DeBERTaV3 model significantly reduces the performance metrics, with the f1 score decreasing drastically when applying it on java-based projects. To determine the efficacy of the model on other projects, a relevant case-study project should be selected, and security issues predicted using the model. Then, the results should be validated together with security experts to determine the number of hidden CVEs. For this case, special attention should be given also to the NSBRs, and at least a random sample should be validated to make sure that SBRs have not been mislabeled.

One of the known limitations of the task of SBR classification is the size and quality of the datasets.

Although the transformer models are great tools to be leveraged, if fine-tuning is done on a low-quality database, the results will be far from satisfactory. The main challenges of the databases are data quality [21], data imbalance [63], and cross-project applicability [25]. For the current research, the problem of data quality has been solved for the large Chromium dataset, as it was manually reviewed by security experts. However, the dataset is still unbalanced, consisting of 808 SBR and 41132 NSBR, and consisting only of C++ issues, which does not account for differences in language patterns and terminology across projects and may reduce the classifier's ability to generalize effectively [52]. Although workarounds for these issues exist, such as the rebalancing methods [63], the dataset for the transformer training can be improved. A suggestion is to gather both SBR and NSBR from different projects to create a more encompassing and diverse dataset. This approach could help increase the performance of the model by allowing it to learn from different sources, and potentially increase cross-project applicability. Alqahtani [52] has investigated with training a FastText classifier on a merged dataset of four projects introduced by Wu et al. [21], and using it to predict SBR on a fifth project. The results on cross-project applicability show an average f1 score of 0.65. FastText uses an older machine-learning model, and results could be improved through the use of more recent and advanced transformer models.

One significant limitation of the validation process is that the results have been validated by only a single expert. This introduces the potential for bias in the validated results, as the assessment and interpretation of security issues are subject to individual expertise and perspectives. Relying on a single expert may lead to inconsistencies or subjective judgments that could skew the findings. To mitigate this bias and enhance the reliability of the validation process, it is essential to involve multiple experts in the evaluation. Having a diverse group of experts would provide a more balanced and comprehensive assessment of security issues, reducing the likelihood of individual biases influencing the outcomes. Additionally, employing inter-rater reliability metrics, such as Krippendorff's alpha, can further ensure consistency and objectivity in the validation process. Krippendorff's alpha measures the agreement among multiple raters, providing a statistical measure of reliability that can highlight any discrepancies and ensure that the validation process is robust and credible [89]. Expanding the validation to include multiple experts and applying rigorous reliability metrics would not only address the bias but also enhance the overall validity and credibility of the findings, leading to more accurate and dependable results.

Concerning the validation of results, it is crucial to underline that only the predicted SBRs have undergone validation for uncovering hidden CVEs. As noted in Section 4.6, it is probable that the NSBRs currently categorized also harbor security issues but have been incorrectly labeled. Unfortunately, the exact number of vulnerabilities misclassified as NSBR, and thus overlooked in the validation process, remains undetermined. The differences between the Chromium and gRPC projects suggest that some vulnerabilities might have been missed [47]. An accurate determination of the total number of concealed vulnerabilities—and thereby a proper assessment of the DeBERTaV3 model's effectiveness—would require a manual verification of each security issue, as described by Wu et al. [21]. However, this approach is notably time-consuming, potentially extending over several months, as highlighted by the authors. Nonetheless, the current validation lends support to our hypothesis of hidden vulnerabilities within issue trackers. Extending this validation to encompass the entire gRPC issue tracker would likely reinforce our findings and provide a more precise measure of the scale of these vulnerabilities.

On the technical front, the process of accurately predicting security labels is hampered by both the limitations in data extraction capabilities of the GitHub API, particularly with respect to code snippets, and the intrinsic limitations of the text-classification model in processing non-textual information like hyperlinks and images. An improvement in recall might be achieved if the issue descriptions were more comprehensive and predominantly text-based. The initial descriptions of issues are often succinct, sometimes comprising only a few words, with the bulk of relevant information emerging through subsequent discussions. Additionally, security labels are frequently applied by developers at a later stage, which means the initial post might lack sufficient details for the model to identify it as a security concern, resulting in a classification as non-security. Issues might contain critical information in forms that are not adequately captured by the GitHub API, such as code snippets, hyperlinks, or images. Code snippets may not be correctly decoded, hyperlinks are overlooked and not followed, and images, which cannot be interpreted by the DeBERTa model due to its focus on text classification, are disregarded during the preprocessing phase. Additionally, an expert validation identified an issue with the

model's handling of interconnected issues. For instance, issue 457 on gRPC, identified as a CVE with a CVSS score of 5.9, is linked to an original issue 429. However, because issue 429 is referenced via a hyperlink, the model lacks the capability to navigate to and interpret content from external URLs, which impedes its ability to assess the issue comprehensively.

Another technical limitation is the hyper-parameter optimization of the DeBERTaV3 model. For the current model, no structured approach has been taken to conduct the hyper-parameter optimization (HPO), mainly as a result of restricted computational resources in the scope of the Masters thesis. HPO has been proven to be a crucial methodology in enhancing the effectiveness of machine learning algorithms, as the performance of the algorithms depends significantly on parametric choices [90]. For example, Algorain & Clark [90] used HPO in a security context, for optimizing a malware detection algorithm. Their work demonstrates through empirical results that HPO leads to significant performance gains over default settings, which often fail to capture the nuances of the dataset effectively. Although in our research the current configuration achieves good results, it is unknown if by applying HPO we could improve the performance of the model, and even circumvent issues such as the precision-recall tradeoff through a better training process.

5.3. Future Work

Building on these findings, future research should dive into understanding the reasons behind developers posting vulnerabilities on issue trackers rather than coordinating with product maintainers and CVE Numbering Authorities. Investigating the motivations and challenges faced by developers in this context is crucial. Such research would be highly relevant in designing targeted policies and interventions aimed at improving the vulnerability disclosure process. Understanding these motivations can help create a more effective framework that encourages responsible disclosure practices and minimizes the risks associated with public exposure of vulnerabilities.

Another important avenue for future research is the investigation of the applicability of fine-tuned transformer models across different programming languages. This research should aim to determine whether fine-tuning a model for each programming language is necessary, or if the knowledge obtained from fine-tuning on one language can be effectively transferred to others. This would significantly enhance the utility of these models in diverse software development environments. Understanding the feasibility of cross-language applications could lead to more efficient and versatile tools for vulnerability detection, thereby broadening the impact of this research.

References

[1] Stuart Millar. “Vulnerability Detection in Open Source Software: An Introduction”. In: *arXiv preprint arXiv:2203.16428* (2022).

[2] Jeferson Martínez and Javier M Durán. “Software supply chain attacks, a threat to global cybersecurity: SolarWinds’ case study”. In: *International Journal of Safety and Security Engineering* 11.5 (2021), pp. 537–545.

[3] G. Schryen. “Security of open source and closed source software: An empirical comparison of published vulnerabilities”. In: *AMCIS 2009 Proceedings*. 2009, p. 387.

[4] Aksel Ethembabaoglu et al. “The Unpatchables: Why Municipalities Persist in Running Vulnerable Hosts”. In: ().

[5] Stephanie de Smale et al. “No one drinks from the firehose: How organizations filter and prioritize vulnerability information”. In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2023, pp. 1980–1996.

[6] *GitLab: Open source end-to-end software development platform with built-in version control, issue tracking, code review, CI/CD, and more*. `https://gitlab.com`. Accessed: 24/05/2024.

[7] *BitBucket: Collaborative Git solution that massively scales*. `https://bitbucket.org`. Accessed: 24/05/2024.

[8] *SourceForge: A web-based service that offers software developers a centralized online location to control and manage free and open-source software projects*. `https://sourceforge.net`. Accessed: 24/05/2024.

[9] *GitHub: The world’s leading software development platform*. `https://github.com`. Accessed: 24/05/2024.

[10] *TensorFlow: An end-to-end open source platform for machine learning*. `https://www.tensorflow.org`. Accessed: 24/05/2024.

[11] *Docker: Empowering App Development for Developers*. `https://www.docker.com`. Accessed: 24/05/2024.

[12] Ashish Arora et al. “Impact of vulnerability disclosure and patch availability-an empirical analysis”. In: *Third workshop on the economics of information security*. Vol. 24. Citeseer. 2004, pp. 1268–1287.

[13] Allen D Householder et al. “The cert guide to coordinated vulnerability disclosure”. In: *Software Engineering Institute (Carnegie Mellon University). Disponible en https://bit. ly/3CSCaz5* (2017).

[14] J. Lin, B. Adams, and A. E. Hassan. “On the coordination of vulnerability fixes: An empirical study of practices from 13 CVE numbering authorities”. In: *Empirical Software Engineering* 28.6 (2023), p. 151.

[15] VAR-IoT Project. *Sharing Data with the World*. Accessed: 2024-06-22. 2024. URL: `https://www.variot.eu/project-outcomes/sharing-data-with-the-world/`.

[16] J. Wachs et al. “The geography of open source software: Evidence from github”. In: *Technological Forecasting and Social Change* 176 (2022), p. 121478.

[17] P. Ladisa et al. “Sok: Taxonomy of attacks on open-source software supply chains”. In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2023, pp. 1509–1526.

[18] Jacob Devlin et al. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).

[19] Yinhan Liu et al. “Roberta: A robustly optimized bert pretraining approach”. In: *arXiv preprint arXiv:1907.11692* (2019).

[20] Pengcheng He, Jianfeng Gao, and Weizhu Chen. “Debertav3: Improving deberta using electra-style pre-training with gradient-disentangled embedding sharing”. In: *arXiv preprint arXiv:2111.09543* (2021).

[21] X. Wu et al. “Data quality matters: A case study on data label correctness for security bug report prediction”. In: *IEEE Transactions on Software Engineering* 48.7 (2021), pp. 2541–2556.

[22] Jeanine Banks and Tris Warkentin. *Gemma: Introducing New State-of-the-Art Open Models*. https://blog.google/technology/developers/gemma-open-models/. Accessed: 24/05/2024. Feb. 2024.

[23] Mistral AI. *Announcing Mistral-7B*. https://mistral.ai/news/announcing-mistral-7b/. Accessed: 24/05/2024. 2023.

[24] *gRPC published vulnerabilities*. https://tinyurl.com/hbax8pxf. Accessed: 24/05/2024. 2024.

[25] S. Chakraborty et al. “Deep learning based vulnerability detection: Are we there yet”. In: *IEEE Transactions on Software Engineering* (2021).

[26] Christine Sund. “Towards an international road-map for cybersecurity”. In: *Online Information Review* 31.5 (2007), pp. 566–582.

[27] Dan Craigen, Nadia Diakun-Thibault, and Randy Purse. “Defining cybersecurity”. In: *Technology innovation management review* 4.10 (2014).

[28] Richard A Kemmerer. “Cybersecurity”. In: *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE. 2003, pp. 705–715.

[29] S. Ransbotham. “An Empirical Analysis of Exploitation Attempts Based on Vulnerabilities in Open Source Software”. In: *Weis*. 2010.

[30] F. Quader and V. P. Janeja. “Insights into organizational security readiness: Lessons learned from cyber-attack case studies”. In: *Journal of Cybersecurity and Privacy* 1.4 (2021), pp. 638–659.

[31] J. Luszcz. “Apache struts 2: how technical and development gaps caused the equifax breach”. In: *Network Security* 1 (2018), pp. 5–8.

[32] A. Khazaei, M. Ghasemzadeh, and V. Derhami. “An automatic method for CVSS score prediction using vulnerabilities description”. In: *Journal of Intelligent & Fuzzy Systems* 30.1 (2016), pp. 89–96.

[33] J. C. Costa et al. “Predicting cvss metric via description interpretation”. In: *IEEE Access* 10 (2022), pp. 59125–59134.

[34] M. C. Sánchez et al. “Software vulnerabilities overview: A descriptive study”. In: *Tsinghua Science and Technology* 25.2 (2019), pp. 270–280.

[35] Y. Li et al. “Open source software security vulnerability detection based on dynamic behavior features”. In: *PLOS ONE* 14.8 (2019), e0221530. DOI: 10.1371/journal.pone.0221530.

[36] M. Zahedi, M. Ali Babar, and C. Treude. “An empirical study of security issues posted in open source projects”. In: *Empirical Software Engineering* (2018).

[37] D. Zou et al. “Automatically identifying security bug reports via multitype features analysis”. In: *Information Security and Privacy: 23rd Australasian Conference, ACISP 2018*. Wollongong, NSW, Australia: Springer International Publishing, 2018, pp. 619–633.

[38] N. Bühlmann and M. Ghafari. “How do developers deal with security issue reports on github?” In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. Apr. 2022, pp. 1580–1589.

[39] F. A. Bhuiyan, M. B. Sharif, and A. Rahman. “Security bug report usage for software vulnerability research: a systematic mapping study”. In: *IEEE Access* 9 (2021), pp. 28471–28495.

[40] D. C. Das and M. R. Rahman. “Security and performance bug reports identification with class-imbalance sampling and feature selection”. In: *2018 Joint 7th International Conference on Informatics, Electronics & Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*. IEEE. 2018, pp. 316–321.

[41] E. Wåreus et al. “Security Issue Classification for Vulnerability Management with Semi-supervised Learning”. In: *ICISSP*. 2022, pp. 84–95.

[42] S. Frei et al. “Large-Scale Vulnerability Analysis”. In: *Symposium on Applied Computing*. 2006.

[43] L. Allodi. “Economic factors of vulnerability trade and exploitation”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1483–1499.

[44] H. Joh and Y. K. Malaiya. “A framework for software security risk evaluation using the vulnerability lifecycle and CVSS metrics”. In: *Proc. International Workshop on Risk and Trust in Extended Enterprises*. 2010, pp. 430–434.

[45] Zhen Li et al. “VulDeePecker: A Deep Learning-Based System for Vulnerability Detection”. In: *arXiv preprint arXiv:1801.01681* (2018).

[46] Noah Ziems and Shaoen Wu. “Security vulnerability detection using deep learning natural language processing”. In: (2021), pp. 1–6.

[47] Michael Gegick, Pete Rotella, and Tao Xie. “Identifying security bug reports via text mining: An industrial case study”. In: (May 2010), pp. 11–20.

[48] Fayola Peters et al. “Text filtering and ranking for security bug report prediction”. In: *IEEE Transactions on Software Engineering* 45.6 (2017), pp. 615–631.

[49] Wei Zheng et al. “A domain knowledge-guided lightweight approach for security bug reports prediction”. In: (Aug. 2021), pp. 359–368.

[50] X. Sun et al. “Automatic software vulnerability assessment by extracting vulnerability elements”. In: *Journal of Systems and Software* (2023), p. 111790.

[51] Y. Zhou and A. Sharma. “Automated identification of security issues from commit messages and bug reports”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Aug. 2017, pp. 914–919.

[52] S. S. Alqahtani. “Security bug reports classification using fasttext”. In: *International Journal of Information Security* (2023), pp. 1–12.

[53] A. Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems*. Vol. 30. 2017.

[54] J. P. Meher, S. Biswas, and R. Mall. “Deep learning-based software bug classification”. In: *Information and Software Technology* 166 (2024), p. 107350.

[55] *Mining Software Repository Challenge*. `http://2011.msrconf.org/msr-challenge.html`. Accessed: insert-access-date-here. 2011.

[56] Xiuting Ge et al. “Locality-based security bug report identification via active learning”. In: *Information and Software Technology* 147 (2022), p. 106899.

[57] Rahul Rastogi and Rossouw von Solms. “Information security governance-a re-definition”. In: *Security Management, Integrity, and Internal Control in Information Systems: IFIP TC-11 WG 11.1 & WG 11.5 Joint Working Conference 7*. Springer. 2005, pp. 223–236.

[58] Roland Croft, M. Ali Babar, and M. Mehdi Kholoosi. “Data quality for software vulnerability datasets”. In: (May 2023), pp. 121–133.

[59] Rui Shu et al. “Better security bug report classification via hyperparameter optimization”. In: *arXiv preprint arXiv:1905.06872* (2019).

[60] Alberto Fernández et al. “SMOTE for learning from imbalanced data: progress and challenges, marking the 15-year anniversary”. In: *Journal of artificial intelligence research* 61 (2018), pp. 863–905.

[61] Nitesh V. Chawla et al. “SMOTE: synthetic minority over-sampling technique”. In: *Journal of Artificial Intelligence Research* 16 (2002), pp. 321–357.

[62] Chakkrit Tantithamthavorn, Ahmed E. Hassan, and Kenichi Matsumoto. “The impact of class rebalancing techniques on the performance and interpretation of defect prediction models”. In: *IEEE Transactions on Software Engineering* 46.11 (2018), pp. 1200–1219.

[63] Yansheng Liao and Tao Zhang. “SEDAC: A CVAE-Based Data Augmentation Method for Security Bug Report Identification”. In: *arXiv preprint arXiv:2401.12060* (2024).

[64] Horácio L. França, César Teixeira, and Nuno Laranjeiro. “An Empirical Analysis of Rebalancing Methods for Security Issue Report Identification”. In: (Oct. 2023), pp. 1–12.

[65] P. Mell, K. Scarfone, and S. Romanosky. “Common vulnerability scoring system”. In: *IEEE Security & Privacy* 4.6 (2006), pp. 85–89.

[66] Thomas Walshe and Andrew C Simpson. “Coordinated vulnerability disclosure programme effectiveness: Issues and recommendations”. In: *Computers & Security* 123 (2022), p. 102936.

[67] TIOBE Software. *TIOBE Index for May 2024*. `https://www.tiobe.com/tiobe-index/`. Accessed: 2024-06-01. 2024.

[68] *React Native*. `https://reactnative.dev/`. Accessed: 26/05/2024.

[69] *OpenCV*. `https://opencv.org/`. Accessed: insert-date-here.

[70] Zhenbo Xu, Jian Zhang, and Zhongxing Xu. “Memory leak detection based on memory state transition graph”. In: *2011 18th Asia-Pacific Software Engineering Conference*. IEEE. 2011, pp. 33–40.

[71] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[72] Patrice Lacroix and Jules Desharnais. “Buffer Overflow Vulnerabilities in C and C++”. In: *Rapport de Recherche DIUL-RR-0803, Université Laval, Québec, Canada* (2008).

[73] Baishakhi Ray et al. “A large scale study of programming languages and code quality in github”. In: *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 2014, pp. 155–165.

[74] A. Dosovitskiy et al. “An image is worth 16x16 words: Transformers for image recognition at scale”. In: *arXiv preprint arXiv:2010.11929* (2020).

[75] T. Brown et al. “Language models are few-shot learners”. In: *Advances in neural information processing systems*. Vol. 33. 2020, pp. 1877–1901.

[76] NVIDIA Corporation. *NVIDIA T4 70W Low Profile PCIe GPU Accelerator*. Product Brief, PB-09256-001_v05. 2020. URL: `https://www.nvidia.com/en-us/data-center/tesla/tesla-qualified-servers-catalog/`.

[77] Anna Nesvijevskaia et al. “The accuracy versus interpretability trade-off in fraud detection model”. In: *Data & Policy* 3 (2021), e12.

[78] Peter Mell, Karen Scarfone, and Sasha Romanosky. “A Complete Guide to the Common Vulnerability Scoring System Version 2.0”. In: *NIST Special Publication* 800-55 (2006). Accessed: 2024-05-16. URL: `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-55.pdf`.

[79] FIRST. *Common Vulnerability Scoring System v3.1: Specification Document*. Accessed: 2024-05-16. 2021. URL: `https://www.first.org/cvss/specification-document`.

[80] Evan Li. *Top 100 C++ Projects on GitHub*. `https://github.com/EvanLi/Github-Ranking/blob/master/Top100/CPP.md`. Accessed: 2023-06-01. 2022.

[81] Xiaoxue Wu. *Improved-sbr-datasets*. `https://github.com/wuxiaoxue/Improved-sbr-datasets`. 2023.

[82] Josh Fruhlinger. *The Heartbleed bug: How a flaw in OpenSSL caused a security crisis*. Accessed: 2024-06-01. 2022. URL: `https://www.csoonline.com/article/562859/the-heartbleed-bug-how-a-flaw-in-openssl-caused-a-security-crisis.html` (visited on 06/01/2024).

[83] *2019 Top 25 CWEs*. `https://cwe.mitre.org/top25/`. Accessed: insert-date-here. 2019.

[84] Robert Byers, David Waltermire, and Christopher Turner. *Collaborative Vulnerability Metadata Acceptance Process (CVMAP) for CVE Numbering Authorities (CNAs) and Authorized Data Publishers*. NIST Interagency/Internal Report (NISTIR) 8246. Gaithersburg, MD: National Institute of Standards and Technology, 2020. URL: `https://doi.org/10.6028/NIST.IR.8246`.

[85] Susan Mitchie, Lou Atkins, and Robert West. *The behaviour change wheel: a guide to designing interventions*. Silverback Publishing, 2014.

[86] Brian Fitzgerald and Klaas-Jan Stol. “Continuous Software Engineering: A Roadmap and Agenda”. In: *Journal of Systems and Software* (2017).

[87] Amutheezan Sivagnanam et al. “On the benefits of bug bounty programs: A study of chromium vulnerabilities”. In: *Workshop on the Economics of Information Security (WEIS)*. 2021.

[88] Muhammad Azeem Akbar et al. “Toward successful DevSecOps in software development organizations: A decision-making framework”. In: *Information and Software Technology* 147 (2022), p. 106894.

[89] Klaus Krippendorff. *Content Analysis: An Introduction to Its Methodology*. Sage Publications, 2019. URL: https://doi.org/10.4135/9781071878781.

[90] Fahad T ALGorain and John A Clark. “Bayesian hyper-parameter optimisation for malware detection”. In: *Electronics* 11.10 (2022), p. 1640.

A

Issue Extraction and Cleaning

```
# -*- coding: utf-8 -*-
"""Issue extraction and cleaning.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/17QO3ZVhgfD5hhlXIS1BBKAQqouZGBakH

Section 0. Installing dependencies
"""

!pip install requests
import requests
import json
import pandas as pd

"""Section 1. Issue extraction

"""

# Configuration
token = ''  # Replace with your actual GitHub token
headers = {'Authorization': 'bearer ' + token}
json_filename = 'github_closed_issues_and_comments.json'

# GraphQL query template for fetching issues and comments with pagination
query_template = """
query ($owner: String!, $repo: String!, $issuesFirst: Int!, $cursor: String) {
  repository(owner: $owner, name: $repo) {
    issues(first: $issuesFirst, after: $cursor, states: CLOSED) {
      edges {
        node {
          number
          title
          body
          comments(first: 100) {
            edges {
              node {
                body
              }
            }
          }
        }
        cursor
      }
      pageInfo {
        endCursor
        hasNextPage
      }
```

```
    }
  }
}
"""
def fetch_rate_limit():
    rate_limit_query = """
    query {
      rateLimit {
        limit
        cost
        remaining
        resetAt
      }
    }
    """
    response = requests.post('https://api.github.com/graphql', json={'query':
        rate_limit_query}, headers=headers)
    if response.status_code == 200:
        rate_limit_data = response.json()['data']['rateLimit']
        return rate_limit_data
    else:
        raise Exception(f"Failed to fetch rate limit status: {response.status_code}")
# Function to fetch issues and comments with GraphQL, including pagination
def fetch_all_closed_issues(owner, repo):
    issues = []
    cursor = None  # Start with no cursor
    hasNextPage = True

    while hasNextPage:
        variables = {
            "owner": owner,
            "repo": repo,
            "issuesFirst": 100,  # Adjust this to fetch more or fewer issues per request
            "cursor": cursor
        }

        response = requests.post('https://api.github.com/graphql', json={'query':
            query_template, 'variables': variables}, headers=headers)
        if response.status_code == 200:
            data = response.json()
            issues_batch = data['data']['repository']['issues']['edges']
            issues.extend(issues_batch)
            pageInfo = data['data']['repository']['issues']['pageInfo']
            hasNextPage = pageInfo['hasNextPage']
            cursor = pageInfo['endCursor']
            print(f"Fetched {len(issues_batch)} issues, total: {len(issues)}")
        else:
            raise Exception(f"Query failed to run by returning code of {response.status_code
                }. {response.text}")

    return [edge['node'] for edge in issues]

def save_to_csv(data, filename):
    df = pd.DataFrame(data)

    # Truncate the 'body' text to the first 10000 characters
    df['body'] = df['body'].apply(lambda x: x[:10000] if isinstance(x, str) else x)

    # Rename columns and save to CSV
    df.rename(columns={'number': 'Issue ID', 'body': 'text'}, inplace=True)
    df.to_csv(filename, index=False, escapechar='\\')  # Add escape character
    print(f"Data saved to {filename}")

def load_json_data(filename):
    with open(filename, 'r', encoding='utf-8') as file:
        return json.load(file)
def main():
    owner = "ClickHouse"      ## Test run for ClickHouse project
    repo = "ClickHouse"       ## Test run for ClickHouse project

    # Check rate limit before fetching data
```

```
    rate_limit_before = fetch_rate_limit()
    print(f"Rate Limit Before: {rate_limit_before['remaining']} / {rate_limit_before['limit']}")

    all_closed_issues = fetch_all_closed_issues(owner, repo)

    with open(json_filename, 'w', encoding='utf-8') as jfile:
        json.dump(all_closed_issues, jfile, ensure_ascii=False, indent=4)

    print(f"All closed issues and their comments have been written to {json_filename}")

    # Load the JSON data
    issues_data = load_json_data(json_filename)

    # Save the relevant data to a CSV file
    csv_filename = 'github_issues.csv'
    save_to_csv(issues_data, csv_filename)

    # Check rate limit after fetching data
    rate_limit_after = fetch_rate_limit()
    print(f"Rate Limit After: {rate_limit_after['remaining']} / {rate_limit_after['limit']}")
    print(f"Rate limit resets at: {rate_limit_after['resetAt']}")

if __name__ == '__main__':
    main()

"""Step 2. Issue cleaning"""

import pandas as pd
import re

# Load your CSV file
df = pd.read_csv('github_issues.csv', encoding="UTF-8", sep=",", engine='python', on_bad_lines='skip')

# Remove rows where any cell is NaN
df = df.dropna()

# Remove rows where any column is empty (after trimming whitespace)
df = df.applymap(lambda x: x.strip() if isinstance(x, str) else x)
df = df[df.astype(str).ne('').all(axis=1)]



# Function to clean the text
def clean_text(text):
    text = text.lower()  # Convert text to lowercase
    text = re.sub(r'<.*?>', '', text)  # Remove HTML tags
    text = re.sub(r'http[s]?://\S+', '', text)  # Remove URLs
    text = re.sub(r'\W', ' ', text)  # Remove special characters
    text = re.sub(r'\s+', ' ', text).strip()  # Remove extra spaces
    return text

# Assuming 'Body' is the column you want to clean, apply the clean_text function
df['text'] = df['text'].astype(str).apply(clean_text)

# Select only the 'Issue ID' and 'cleaned_text' columns for saving
columns_to_save = df[['Issue ID', 'text']]

# Save the selected columns to a new CSV file
columns_to_save.to_csv('pre-processed_issues.csv', sep=",", index=False)
```

B

Model finetuning and prediction

```
# -*- coding: utf-8 -*-
"""Fine-tuning and predicting.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1SszyWuOq5pneNvGDqlkfPJy2bG4_uTfO

Section 0. Installing dependencies
"""

!pip install -q -U accelerate bitsandbytes datasets peft transformers tokenizers
    sentencepiece wandb nlpaug
import pandas as pd
import datasets
from transformers import DebertaV2Tokenizer, DebertaV2ForSequenceClassification,Trainer,
    TrainingArguments,AutoTokenizer
import torch.nn as nn
import torch
from torch.utils.data import Dataset, DataLoader
import numpy as np
from sklearn.metrics import accuracy_score, precision_recall_fscore_support, confusion_matrix
from tqdm import tqdm
import wandb
import os
from datasets import load_dataset
from transformers import EarlyStoppingCallback
from torch.nn.functional import softmax
from tqdm.auto import tqdm

"""Section 1 Model fine-tuning"""

eval_results = pd.DataFrame()  ## Creating variable to store performance metrics, only has to
     be run once

# load model and tokenizer and define length of the text sequence
model = DebertaV2ForSequenceClassification.from_pretrained("microsoft/deberta-v3-base")
tokenizer = DebertaV2Tokenizer.from_pretrained("microsoft/deberta-v3-base", max_length = 512)
dataset = load_dataset("csv", data_files='sample_data/Chromium-TRAIN-preprocessed.csv',
    encoding='UTF-8')

dataset_split = dataset['train'].train_test_split(test_size=0.2, seed=42)
train_data = dataset_split['train']
test_data = dataset_split['test']

def tokenization(batched_text):
    return tokenizer(batched_text['text'], padding='max_length', truncation=True, max_length
        = 512)

```

```
train_data = train_data.map(tokenization, batched = True, batch_size = len(train_data))
test_data = test_data.map(tokenization, batched = True, batch_size = len(test_data))

train_data.set_format('torch', columns=['input_ids', 'attention_mask', 'label'])
test_data.set_format('torch', columns=['input_ids', 'attention_mask', 'label'])

# define accuracy metrics
def compute_metrics(pred):
    labels = pred.label_ids
    probs = pred.predictions[:,1]  # Assuming the second column represents the "positive" class probabilities
    threshold = 0.7  # Example threshold, adjust based on your needs
    preds = (probs > threshold).astype(int)
    precision, recall, f1, _ = precision_recall_fscore_support(labels, preds, average='binary')
    acc = accuracy_score(labels, preds)
    cm = confusion_matrix(labels, preds)
    print("Confusion Matrix:")
    print(cm)
    return {
        'accuracy': acc,
        'f1': f1,
        'precision': precision,
        'recall': recall
    }

training_args = TrainingArguments(
    output_dir ="fine_tuned_deberta",
    num_train_epochs=10,
    # eval_steps=500,
    per_device_train_batch_size = 8,
    gradient_accumulation_steps = 8,
    per_device_eval_batch_size= 8,
    evaluation_strategy = "epoch",
    save_strategy="epoch",  # Save the model at the same frequency as evaluations
    # save_steps=500,  # Save the model every 500 steps, matching eval_steps
    disable_tqdm = False,
    warmup_steps=100,
    weight_decay=0.01,
    logging_steps = 8,
    fp16 = True,
    logging_dir='content/',
    dataloader_num_workers = 8,
    push_to_hub = True,
    push_to_hub_token = "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",
    run_name = 'roberta-classification',
    load_best_model_at_end=True,  # Optional but useful: load the best model at the end of training
    metric_for_best_model='accuracy',  # Specify which metric to use for early stopping
)

# instantiate the trainer class and check for available devices
trainer = Trainer(
    model=model,
    args=training_args,
    compute_metrics=compute_metrics,
    train_dataset=train_data,
    eval_dataset=test_data,
    callbacks=[EarlyStoppingCallback(early_stopping_patience=2)],
  )
device = 'cuda' if torch.cuda.is_available() else 'cpu'
device

trainer.train()


results_eval = trainer.evaluate()
eval_df = pd.DataFrame([results_eval])
eval_results = pd.concat([eval_results, eval_df], ignore_index=True)

eval_results
```

```

test_results = pd.DataFrame()  ## Creating variable to store performance metrics, only has to be run once

new_dataset = load_dataset("csv", data_files='sample_data/Chromium-TEST-preprocessed.csv', encoding='UTF-8')
test_data = new_dataset.map(tokenization, batched = True, batch_size = len(train_data))
test_data.set_format('torch', columns=['input_ids', 'attention_mask', 'label'])
results_test = trainer.evaluate(test_data)
test_df = pd.DataFrame([results_test])
test_results = pd.concat([test_results, test_df], ignore_index=True)
test_results

"""Section 2 Model testing with loading from huggingface"""

# Commented out IPython magic to ensure Python compatibility.
!git clone https://huggingface.co/Bugsec/fine_tuned_deberta
# %cd fine_tuned_deberta
!git checkout 51865f0cfb7324c3e00ba3054acfb53015edb6f8

model = DebertaV2ForSequenceClassification.from_pretrained('.')
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
tokenizer = DebertaV2Tokenizer.from_pretrained("microsoft/deberta-v3-base")
# %cd ..
def tokenization(batched_text):
    return tokenizer(batched_text['text'], padding='max_length', truncation=True, max_length = 512)
new_dataset = load_dataset("csv", data_files='sample_data/Chromium-TEST-preprocessed.csv', encoding='UTF-8')
test_data = new_dataset.map(tokenization, batched = True, batch_size = 1000)
test_data.set_format('torch', columns=['input_ids', 'attention_mask', 'label'])
from torch.utils.data import DataLoader
from tqdm.auto import tqdm
from torch.nn.functional import softmax
import torch

# Assuming the model and device setup from your previous snippets
model.eval()

# Create DataLoader
test_dataloader = DataLoader(test_data['train'], batch_size=16)  # Adjust the batch size as needed

# Initialize lists for true labels and predictions
true_labels = []
pred_labels = []
probs_list = []
# Define the custom threshold
custom_threshold = 0.50

# Evaluation loop
for batch in tqdm(test_dataloader, desc="Evaluating"):
    batch = {k: v.to(device) for k, v in batch.items()}
    labels = batch.pop('label').cpu().numpy()  # Correctly reference the label
    true_labels.extend(labels)

    with torch.no_grad():
        outputs = model(**batch)
        logits = outputs.logits
        probs = softmax(logits, dim=1)
        probs_list.extend(probs.cpu().numpy())
        preds = (probs[:, 1] > custom_threshold).long().cpu().numpy()
        pred_labels.extend(preds)

# Calculate metrics
from sklearn.metrics import accuracy_score, precision_recall_fscore_support
precision, recall, f1, _ = precision_recall_fscore_support(true_labels, pred_labels, average='binary')
accuracy = accuracy_score(true_labels, pred_labels)

```

```
# Output results
print(f"Accuracy: {accuracy:.4f}, Precision: {precision:.4f}, Recall: {recall:.4f}, F1: {f1:.4f}")

"""Section 2.1 Model results threshold visualisation"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Convert the lists to numpy arrays for easier indexing
true_labels_array = np.array(true_labels)
pred_labels_array = np.array(pred_labels)
probs_array = np.array(probs_list)[:, 1]  # Assuming index 1 is the 'security' class

# Define bins for probabilities
bins = np.linspace(0.5, 1, 11)  # 20 bins between 0 and 1
bin_labels = [f"{bins[i]:.2f}-{bins[i+1]:.2f}" for i in range(len(bins)-1)]

# Create a DataFrame with the probabilities and their true/predicted labels
df = pd.DataFrame({
    'Probability': probs_array,
    'True Label': true_labels_array,
    'Predicted Label': pred_labels_array
})

# Categorize probabilities into bins
df['Probability Bin'] = pd.cut(df['Probability'], bins, labels=bin_labels, include_lowest=True)

# Calculate counts for each scenario
predicted_positives = df[(df['Probability'] >= 0.5) & (df['Predicted Label'] == 1)]['Probability Bin'].value_counts().reindex(bin_labels, fill_value=0)
true_positives = df[(df['True Label'] == 1) & (df['Predicted Label'] == 1)]['Probability Bin'].value_counts().reindex(bin_labels, fill_value=0)

# Combine into a single DataFrame
stacked_data = pd.DataFrame({
    'Predicted Positives': predicted_positives,
    'True Positives': true_positives
})

# Plotting
ax = stacked_data.plot.bar(stacked=True, figsize=(14, 6), color=['red', 'green'])
plt.title('Distribution of Prediction Probabilities for Positives')
plt.xlabel('Probability Range')
plt.ylabel('Count')
plt.legend(title='Categories')
plt.show()

"""Section 3. **Predicting security labels for projects**"""

!huggingface-cli login

# Commented out IPython magic to ensure Python compatibility.
# Assuming the code to download and set up the model has been run as shown in your testing code
# Git operations and model initialization as per your first snippet
!git clone https://huggingface.co/Bugsec/fine_tuned_deberta
# %cd fine_tuned_deberta
!git checkout 51865f0cfb7324c3e00ba3054acfb53015edb6f8

model = DebertaV2ForSequenceClassification.from_pretrained('.')
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
tokenizer = DebertaV2Tokenizer.from_pretrained("microsoft/deberta-v3-base")
# %cd ..

# Load and tokenize the dataset
def tokenization(batched_text):
    return tokenizer(batched_text['text'], padding='max_length', truncation=True, max_length
```

```
        =512)

predict_dataset = load_dataset("csv", data_files='sample_data/ClickHouse.csv', encoding='UTF-8')  ## An example dataset obtained from extracted issues
predict_dataset = predict_dataset.map(tokenization, batched=True)
predict_dataset.set_format('torch', columns=['input_ids', 'attention_mask'])

# Create DataLoader
predict_dataloader = DataLoader(predict_dataset['train'], batch_size=50, num_workers=32)

# Prediction
model.eval()  # Put the model in evaluation mode
preds = []
custom_threshold = 0.95  # Define the custom threshold as in your test code
for batch in tqdm(predict_dataloader, desc="Predicting"):
    batch = {k: v.to(device) for k, v in batch.items()}  # Move batch to device
    with torch.no_grad():
        outputs = model(**batch)
        logits = outputs.logits
        probs = softmax(logits, dim=1)
        # Use the custom threshold to determine labels
        pred_labels = (probs[:, 1] > custom_threshold).long()
        preds.extend(pred_labels.cpu().tolist())

# Mapping from model output to label
label_dict = {0: 'non-security', 1: 'security'}

# Load the original CSV file for merging predictions
df = pd.read_csv('sample_data/ClickHouse.csv', sep=',')

# Ensure preds array matches the number of rows in df
full_preds = [label_dict[pred] for pred in preds]

# Add the predictions as a new column to the dataframe
df['predictions'] = full_preds

# Save the updated DataFrame to a new CSV file
df.to_csv('sample_data/ClickHouse_cleaned_with_predictions.csv', index=False)

print("CSV␣file␣has␣been␣updated␣with␣predictions.")
```