DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN COMPUTER ENGINEERING

"EVALUATION OF LLM-BASED TOOLS FOR THE AUTOMATED REPAIR OF SECURITY VULNERABILITIES IN ANDROID APPS"

Relatore: Prof. / Dott LOSIOUK ELEONORA

Laureando/a: Saad Ahmed

ANNO ACCADEMICO 2023 – 2024

Data di laurea 04/04/2024

Acknowledgement

I would like to express my heartfelt gratitude to Professor Eleonora, whose unwavering support, invaluable guidance, and profound expertise have played a pivotal role in shaping my academic journey. Her dedication to excellence and passion for imparting knowledge have inspired me to strive for greatness in my studies. I am profoundly grateful for her mentorship and encouragement throughout the course of this thesis.

Abstract

With the rapid increase in Android apps, strong security measures are needed to protect against potential weaknesses. This research study evaluates three tools based on Language Models (LLMs) - ChatGPT, Google Bard, and Android Studio Bot - for automatically fixing security vulnerabilities. To ensure accuracy, the evaluation uses a database of 80 vulnerable code snippets from Google Android Security Bulletins, the study employs BLEU scores to assess the accuracy of code repairs and supplements this with human evaluation to compare fixed codes with correct ones.Among the three large language models (LLMs) evaluated for fixing Android vulnerabilities, ChatGPT outperformed the others by correcting the most vulnerabilities (41 out of 80). While Android Studio Bot achieved a higher BLEU score, indicating syntactic similarity to correct code, it lacked logical correctness.

In summary, this study delves into the potential of LLM-based tools for automatically fixing Android security vulnerabilities. The suggestions and outcomes provided by these tools offer valuable insights, aiding us in determining their effectiveness. This advancement fosters the adoption of automated security measures in Android app development.

Contents

List of Figures xi

List of Tables xiii

List of Code Snippets xvii

1 Introduction 1
- 1.1 Objectives . . . 3

2 Background 5
- 2.1 Software vulnerabilities . . . 5
- 2.2 Common Techniques for Software Vulnerability Repair . . . 5
  - 2.2.1 Static and Dynamic Analysis . . . 5
  - 2.2.2 Fuzzing . . . 6
  - 2.2.3 Machine Learning-Driven Approaches . . . 6
- 2.3 Challenges in Android Vulnerability Repair . . . 6
- 2.4 Introduction to LLM-Based Tools . . . 7
  - 2.4.1 How Large Language Model Works . . . 8
- 2.5 Principles and Mechanisms of LLM-Based Automated Repair . . . 8
- 2.6 Relevance of LLM-Based Approach in Android Security . . . 10
- 2.7 Role of LLMs in Vulnerability Repair . . . 11

3 Related Work 13

4 Methodology 21
- 4.1 Research Design . . . 21
- 4.2 Data Collection . . . 21
- 4.3 Tool Selection . . . 27
- 4.4 Prompt . . . 29
  - 4.4.1 Prompt Used . . . 29
- 4.5 Pre-Processing Phase . . . 30
  - 4.5.1 Pre-processing of LLM Model outputs . . . 30
- 4.6 Evaluation Metrics . . . 30
- 4.7 Experimental Setup . . . 31
- 4.8 Data Analysis Techniques . . . 32

5 Results **33**
5.1 Results Overview 33
5.2 Evaluation Matrix 34
5.2.1 Code Bleu 35
5.2.2 Results Table 38
5.2.3 Human Evaluation Matrix 40
5.3 Overview of Results 42
5.3.1 Quantitative Metrics 42

6 Conclusions and Future Works **45**
6.1 Conclusion 45
6.2 Future Direction 46

7 Appendix **49**
7.1 Example 1 49
7.2 Example 2 52
7.3 Example 3 55

References **61**

63

List of Figures

2.1 Transformer Architecture 9

4.1 Research Diagram 22
4.2 Data Available for a Specific Vulnerability 23
4.3 Format of Availability of Buggy Code and Correct Code for a Specific Vulnerability 24
4.4 Parameters stored in the database of Vulnerabilities 26
4.5 Example of Prompt Used to generate result from LLM Models . . 30
4.6 Proposed fix for Android specific vulnerability by ChatGPT. . . . 31

5.1 CodeBlue weighted N-Gram match [17]. 36
5.2 CodeBlue syntactic AST match [17] 36
5.3 CodeBlue semantic data flow [17] 37
5.4 Graphic Representation of Overall BlueScore of three LLM Models 41
5.5 Graphic Representation of Overall Human Evaluation of three LLM Models 42
5.6 Overall Evaluation Result Correct fix vulnerable code 43

List of Tables

4.1 Dataset parameters 24

5.1 Evaluation Metrics for evaluating CodeBLEU of ChatGPT-3.5 for Android Java code vulnerabilities 38
5.2 Evaluation Metrics for evaluating CodeBLEU of Google Bard for Android Java code vulnerabilities 38
5.3 Evaluation Metrics for evaluating CodeBLEU of Android Studio Bot for Android Java code vulnerabilities 39
5.4 Average of overall result 40
5.5 Human Evaluation Results 41
5.6 Overall Evaluation Results 43

List of Code Snippets

7.1 Vulnerable code of CV A-242703780 50
7.2 Correct code of CV A-242703780 50
7.3 Fix Generated by ChatGPT of CV A-242703780 50
7.4 Fix Generated by Android Studio Bot of CV A-242703780 52
7.5 Fix Generated by Google bard of CV A-242703780 52
7.6 Vulnerable code of CVE-20222489 53
7.7 Correct code of CVE-20222489 53
7.8 Fix Generated by ChatGPT of CVE-20222489 53
7.9 Fix Generated by Android Studio Bot of CVE-20222489 54
7.10 Fix Generated by Google Bard of CVE-20222489 54
7.11 Vulnerable code of CVE-202144228 55
7.12 Correct code of CVE-202144228 56
7.13 Fix Generated by ChatGPT of CVE-202144228 56
7.14 Fix Generated by Android Studio Bot of CVE-202144228 58
7.15 Fix Generated by Google Bard of CVE-202144228 58

Introduction

Android maintained its position as the leading mobile operating system worldwide in the fourth quarter of 2023 with a market share of 70.1% [20]. Android utility tendencies are developed using the Java or Kotlin programming languages and run on the Android Runtime (ART) or the Dalvik Virtual Machine (DVM). Android packages are disbursed through various structures, which includes the Google Play Store, the Amazon App Store, Samsung Galaxy Store, or third-party web sites.

The popularity and variety of Android applications also pose full-size safety challenges, as they may incorporate vulnerabilities that may be exploited through malicious actors. According to a recent study conducted by Check Point Research, it was found that 38% of Android applications analyzed had critical vulnerabilities in 2023 [18]. Some of the common place varieties of Android vulnerabilities consist of insecure facts storage, inadequate encryption, code injection, privilege escalation, and denial-of-carrier. Users and builders of Android applications alike may also suffer grave repercussions from these vulnerabilities, which includes money losses, identification theft, information breaches, reputational damage, and legal ramifications. As a result, it is crucial to identify and address these vulnerabilities as quickly as possible to prevent attackers from exploiting them.

Manual vulnerability detection and repair can be time-consuming, high-priced, and errors-susceptible, mainly for massive and complex applications. Moreover, the developers may not have sufficient expertise or cognizance of the exceptional protection practices or may also forget about a few subtle or hidden vulnerabilities of their code. Therefore, there is a need for automated process to help developers in identifying and resolving protection vulnerabilities in Android programs.

One of the promising tactics for automatic vulnerability repair is the usage of Language Models (LLMs), synthetic neural networks, that can analyze the statistical styles and regulations of natural or programming languages from

massive corpora of text. LLMs can generate herbal-sounding textual content or code snippets based totally on a given context or prompt, and can also perform diverse duties including textual content summarization, translation, class, and finishing touch.

Several LLM-primarily based gear have been proposed or advanced for computerized vulnerability restore in Android programs, consisting of ChatGPT, Google Bard, and Android Studio Bot. These equipment leverage the strength of LLMs to generate guidelines for solving susceptible code snippets, based totally at the description or the severity of the vulnerability. However, the effectiveness and reliability of these tools have never been very well evaluated or compared, and their obstacles and challenges have now not been nicely-understood.

This thesis aims to fill this gap by engaging in a comprehensive evaluation of 3 LLM-based tools for computerized Android safety vulnerability repair: ChatGPT, Google Bard, and Android Studio Bot. The evaluation is based on a database of 80 vulnerable code snippets sourced from Google Android Security Bulletins, which give respectable records and patches for safety vulnerabilities in Android devices. The evaluation technique involves recording the consequences of vulnerability fixes generated through each LLM-based device, and comparing them with the actual fixes furnished with the aid of Google. Two renowned assessment techniques are hired: first, the calculation of BLEU ratings to quantify the syntactic and semantic correctness of the maintenance, and second, manual human evaluation for a more nuanced assessment. The outcomes of the assessment offer insights into the strengths and boundaries of each LLM-based tool concerning syntactic and semantic accuracy, and depicts the strong and weak areas. The combination of automated evaluation metrics and human assessment provides depth to the evaluation, enhancing the reliability of the findings.

1.1 Objectives

The purpose of this research is structured to effectively address the identified problems focused around following key notions.

1. Prompting LLMs for repairing Android vulnerabilities collected from the Bulletins.
2. Evaluating the quality of the repairs through both manual validation and standard metrics.

2 Background

2.1 Software vulnerabilities

Software vulnerabilities pose significant risks to data security, privacy, and system integrity. Addressing these vulnerabilities requires robust techniques and tools for detection and repair. In this chapter, we provide a comprehensive background on software vulnerability repair, focusing on Android applications. We begin by discussing common techniques employed in vulnerability repair and the unique challenges posed by the Android ecosystem. We then explore the role of Language Models (LLMs) in automating vulnerability repair tasks and discuss evaluation metrics used to assess the effectiveness of security tools.

2.2 Common Techniques for Software Vulnerability Repair

Software vulnerability repair involves identifying and remedying weaknesses in software systems to enhance security and reliability. Various techniques have been developed for this purpose, each with its strengths and limitations.

2.2.1 Static and Dynamic Analysis

Static analysis involves examining source code or binary executable without executing the program. It aims to identify potential vulnerabilities through code inspection and pattern matching techniques. While static analysis tools are effective in detecting known vulnerabilities and common coding errors, they may produce false positives and struggle with complex vulnerabilities.

Dynamic analysis, on the other hand, analyzes software behavior during runtime by monitoring program execution and input/output interactions. This approach provides more accurate results compared to static analysis but may incur higher performance overhead.

2.2.2 Fuzzing

Fuzzing is a testing technique that involves systematically feeding invalid, unexpected, or random inputs to a software application to uncover vulnerabilities. Fuzzing tools, such as AFL (American Fuzzy Lop) and libFuzzer, have been highly effective in identifying security vulnerabilities, including memory corruption bugs and input validation flaws.

2.2.3 Machine Learning-Driven Approaches

Machine learning-driven approaches leverage algorithms to analyze data and generate remediation strategies automatically. These approaches hold promise for enhancing the accuracy and efficiency of vulnerability detection and repair. However, they also present challenges related to model interpretability, adversarial attacks, and data privacy.

2.3 Challenges in Android Vulnerability Repair

Repairing vulnerabilities in Android applications presents unique challenges due to the platform's diversity and openness. The Android ecosystem encompasses a wide range of devices, operating system versions, and application types, contributing to a diverse set of vulnerabilities. Common challenges in Android vulnerability repair include:

Input Validation Flaws: Android applications often fail to properly validate user inputs, leading to vulnerabilities such as SQL injection attacks and authentication bypass.

Privilege Escalation Issues: Flaws in the Android permission model can allow malicious applications to elevate their privileges and gain unauthorized access to sensitive resources.

Cryptographic Weaknesses: Improper implementation or misuse of cryptographic algorithms can lead to vulnerabilities such as weak encryption and insecure key management.

Insecure Data Storage: Many Android applications store sensitive information locally on the device without adequate protection, leaving it vulnerable to unauthorized access.

Addressing these challenges remains an open research problem, requiring innovative solutions to ensure the security and privacy of Android users.

2.4 Introduction to LLM-Based Tools

Language Model (LLM)-based tools represent a cutting-edge approach to automated software development tasks, including code generation, summarization, translation, and more recently, security vulnerability repair. LLMs are large-scale machine learning models trained on vast amounts of text data, enabling them to understand and generate human-like text based on given inputs. These models, often based on architectures like GPT (Generative Pre-trained Transformer), have demonstrated remarkable capabilities in natural language understanding and generation, making them suitable for a wide range of natural language processing (NLP) tasks.

In the context of automated security vulnerability repair in Android applications, LLM-based tools leverage their language understanding capabilities to analyze vulnerable code snippets and generate corresponding fixes or suggestions. These tools typically operate by processing natural language descriptions of security vulnerabilities and generating code patches or modifications tailored to address the identified issues. By learning from large datasets of code and textual descriptions, LLM-based tools can infer patterns and relationships between vulnerabilities and their corresponding fixes, enabling them to propose effective solutions autonomously.

One of the key advantages of LLM-based tools is their ability to understand contextual information and generate human-readable code snippets that align with established coding conventions and best practices. This enables to integrate the suggested fixes seamlessly into the codebase, minimizing the need for manual intervention and accelerating the vulnerability remediation process. Additionally, LLM-based tools can adapt to diverse programming languages and coding styles, making them versatile and applicable across various software development environments.

However, LLM-based tools are not without limitations. Challenges such as model interpretability, bias, and robustness to adversarial inputs can impact the reliability and effectiveness of these tools, particularly in security-critical scenarios. Furthermore, the quality of code patches generated by LLM-based tools may vary depending on factors such as the complexity of the vulnerability, the clarity of the vulnerability description, and the size of the training dataset used to fine-tune the model.

In summary, LLM-based tools represent a promising paradigm for automated security vulnerability repair in Android applications. Their ability to leverage natural language understanding and generation capabilities offers significant potential for improving the efficiency and effectiveness of vulnerability remediation processes. However, addressing the challenges associated with model interpretability, bias, and robustness will be essential to realizing the full potential of LLM-based tools in enhancing Android application security.

2.4.1 How Large Language Model Works

LLMs rely on machine learning methodologies, leveraging deep learning and vast datasets. One foundational structure widely adopted in LLMs is the transformer architecture. Lets explore this architecture in more detail:

Transformer Architecture The transformer is a deep learning architecture developed by Google, introduced in the influential paper Attention Is All You Need in 2017 [22]. Unlike traditional recurrent neural networks (RNNs), transformers do not rely on recurrence. Instead, they use an attention mechanism to draw global dependencies between input and output.

Here are the key components of the transformer:

Embedding: Converts one-hot encoded tokens into vectors representing the tokens. **Stack of Encoders**: These are the transformer encoders that perform transformations over the array of representation vectors. **Un-embedding**: Converts the final representation vectors back into one-hot encoded tokens. While necessary for pre-training, its often unnecessary for downstream tasks.

The transformer architecture has been instrumental in advancing language-centric AI systems. Its used not only in natural language processing but also in computer vision, audio, and multi-modal processing [5]. The transformer architecture is illustrated in Figure 2.1.

2.5 Principles and Mechanisms of LLM-Based Automated Repair

LLM-based automated repair systems operate on the principles of natural language processing (NLP) and machine learning (ML), leveraging large-scale language models to understand, interpret, and generate code snippets that address identified security vulnerabilities. These systems are typically based on transformer architectures, such as GPT (Generative Pre-trained Transformer), which have demonstrated exceptional performance in various NLP tasks.

The mechanism of LLM-based automated repair involves several key steps:

Input Processing: The system receives input in the form of natural language descriptions of security vulnerabilities, often sourced from security bulletins, bug reports, or other documentation. These descriptions may include details such as the type of vulnerability, affected code snippets, and potential impacts.

Contextual Understanding: The LLM-based model processes the input text to extract contextual information relevant to the security vulnerability. By analyzing the semantics and syntax of the input, the model gains an understanding of the underlying issue and its implications for the codebase.

Code Generation: Based on the contextual understanding acquired from the input text, the LLM-based model generates code snippets that propose fixes or mitigation's for the identified vulnerability. These code snippets are designed to address the specific security concerns outlined in the input description while

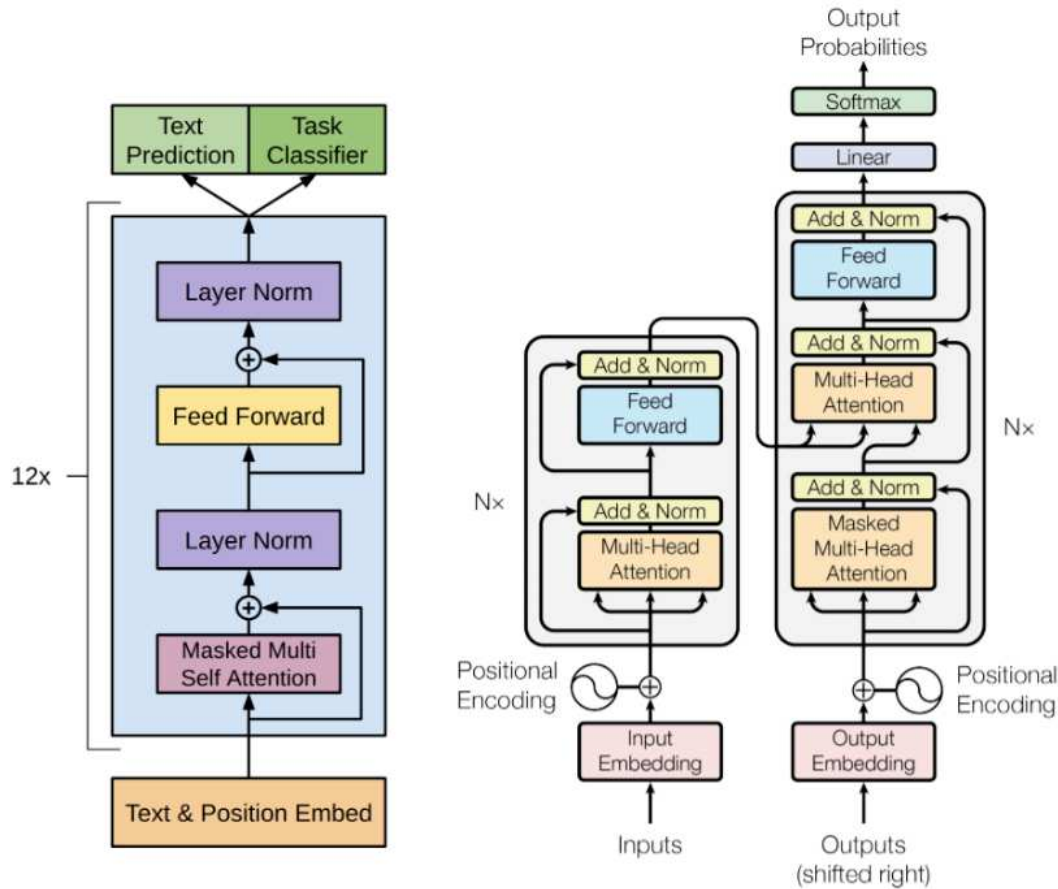

Figure 2.1: Transformer Architecture
[21]

adhering to coding conventions and best practices.

Quality Assessment: The generated code snippets undergo evaluation to assess their correctness, effectiveness, and adherence to security guidelines. This assessment may involve automated testing, code review by human experts, or comparison against known patches or fixes provided by software vendors or security researchers.

Feedback Loop: Feedback from the evaluation process is used to refine and improve the performance of the LLM-based repair system. This feedback loop may involve fine-tuning the model on labeled data, adjusting generation parameters, or incorporating domain-specific knowledge to enhance the system's capabilities over time.

The effectiveness of LLM-based automated repair systems relies on the quality of the language model, the relevance and accuracy of the input descriptions, and the robustness of the code generation process. Additionally, factors such as the diversity and representatives of the training data, the size of the model architecture, and the availability of computational resources can influence the

performance of these systems.

Overall, LLM-based automated repair systems represent a novel and promising approach to addressing security vulnerabilities in software applications. By leveraging advanced NLP and ML techniques, these systems have the potential to streamline the vulnerability remediation process, reduce manual effort, and improve the overall security posture of software systems. However, continued research and development efforts are needed to address challenges such as model interpretability, bias mitigation, and the integration of human feedback into the repair process.

2.6 Relevance of LLM-Based Approach in Android Security

The relevance of Language Model (LLM)-based approaches in Android security stems from the need for efficient and effective methods to detect and mitigate security vulnerabilities in mobile applications. As the Android ecosystem continues to expand, with millions of applications available on the Google Play Store, ensuring the security and integrity of these applications is paramount to protect user data and maintain user trust.

LLM-based approaches offer several advantages that make them particularly well-suited for addressing security challenges in the Android environment:

Semantic Understanding: LLMs, such as GPT (Generative Pre-trained Transformer), are trained on vast amounts of text data, enabling them to understand and interpret natural language descriptions of security vulnerabilities. In the context of Android security, where vulnerability reports and security bulletins often contain textual descriptions of issues, LLMs can analyze these descriptions to identify relevant vulnerabilities and propose appropriate fixes.

Code Generation Capabilities: LLMs are capable of generating code snippets based on natural language prompts, allowing them to propose fixes for identified security vulnerabilities in Android applications. This code generation process can significantly accelerate the vulnerability remediation process, especially for common security issues such as input validation flaws, cryptographic weaknesses, and privilege escalation vulnerabilities.

Adaptability to Android Development Practices: LLM-based approaches can be tailored to the specifics of Android application development, including the use of Java and Kotlin programming languages, the Android SDK (Software Development Kit), and common development frameworks such as Android Studio. By training LLMs on datasets specific to Android development practices and security considerations, these approaches can generate more accurate and relevant code suggestions for Android security vulnerabilities.

Scalability and Automation: LLM-based approaches have the potential to scale across large numbers of Android applications and security vulnerabilities, automating much of the vulnerability detection and repair process. This

scalability is particularly important given the sheer volume of Android applications and the rapid pace of security research and disclosures in the Android ecosystem.

Integration with Development Workflow: LLM-based tools can be integrated into existing development workflows and tools used by Android developers, such as IDEs (Integrated Development Environments) and version control systems. This integration streamlines the vulnerability remediation process, allowing developers to quickly apply suggested fixes and incorporate security best practices into their codebase.

Overall, the relevance of LLM-based approaches in Android security lies in their ability to leverage natural language understanding and code generation capabilities to automate and expedite the detection and mitigation of security vulnerabilities in Android applications. As the Android ecosystem continues to evolve, LLM-based approaches offer a promising avenue for enhancing the security posture of mobile applications and protecting users from potential threats.

2.7 Role of LLMs in Vulnerability Repair

Language Model (LLM)-based approaches have emerged as promising solutions for automating vulnerability repair tasks. LLMs, such as GPT (Generative Pre-trained Transformer) and BERT (Bidirectional Encoder Representations from Transformers), possess the ability to understand and generate natural language text, making them suitable for analyzing code snippets and generating patches.

By leveraging LLMs in vulnerability repair, it becomes possible to identify and remediate security vulnerabilities more efficiently and accurately. These approaches hold potential for improving the security posture of software systems, including Android applications, by automating the detection and repair of vulnerabilities.

Related Work

Automated test generation techniques have proven invaluable for developers in writing tests, with a primary focus on increasing coverage or generating exploratory input. However, a critical yet often overlooked aspect is the generation of tests specifically designed to reproduce reported bugs in software. Empirical research indicates that a significant proportion of tests added to open source repositories originate from bug reports. Existing methods for error reconstruction typically focus on crashing activities, overlooking the diverse range of reported events. In response to this gap, LIBRO, a framework leveraging large language models (LLMs), is proposed for test generation based on common error reports. While LLMs themselves cannot directly generate buggy code, post-processing steps are employed to verify efficiency and compare test results for accuracy. Analysis of the Defects4J benchmark demonstrates LIBRO's ability to reproduce failures for a significant portion of the studied cases, suggesting a prioritization of virus-breeding testing. Furthermore, concerns regarding data pollution are mitigated by evaluating LIBRO against bug reports collected after LLM training, showcasing its potential to enhance developer productivity through test generation from bug reports [12].

Automated Program Repair (APR) techniques are pivotal in enhancing software reliability by automatically generating patches for faulty programs. However, recent advancements in deep learning (DL) approaches encounter challenges stemming from the abundance of syntactically or semantically ambiguous terms within the patch space. These challenges often lead to deviations from the syntactic and semantic constraints of the source code, thereby hindering the generation of accurate solutions. In response to this challenge, KNOD, a DL-based APR method, is proposed to integrate domain knowledge for effectively guiding patch generation. KNOD introduces two key innovations: (1) a novel three-step tree decoder designed to directly produce Abstract Syntax Trees of patched code, aligning with the underlying tree structure, and (2) a new domain-rule distillation mechanism that incorporates syntax and semantic rules into the training and reasoning phases. This mechanism influences the decoding process through teacher and student classification. KNOD is evaluated on three widely-used benchmarks to showcase its efficacy in error prevention across diverse datasets when compared to existing APR tools [11].

Automated Program Repair (APR) aims to bolster software reliability by automatically generating patches for flawed programs. Despite the demonstrated effectiveness of various Code Language Models (CLMs) in tasks such as code completion, their potential for APR tasks remains relatively underexplored. This paper conducts a thorough investigation into the repair capabilities of CLMs for APR tasks. Initially, ten CLMs are evaluated across four APR benchmarks, revealing that the top-performing CLM, in its original state, successfully rectifies a noteworthy 72% more bugs compared to state-of-the-art deep learning (DL)-based APR techniques. Moreover, a novel APR benchmark is introduced to ensure a fair evaluation without data leakage. Furthermore, the paper pioneers the fine-tuning of CLMs with APR training data, showcasing substantial improvements ranging from 31% to 1,267% over existing DL-based APR techniques, resulting in the resolution of 46% to 164% more bugs. The study also delves into the impact of buggy lines, uncovering that while CLMs in their original state struggle to effectively utilize buggy lines, fine-tuned CLMs may potentially over-rely on them. Lastly, the efficiency metrics, including size, time, and memory, of different CLMs are analyzed. This work illuminates promising avenues for advancing the APR domain, underlining the significance of fine-tuning CLMs with APR-specific designs and advocating for transparent reporting of open-source repositories to mitigate data leakage issues [10].

When confronted with unintended application behavior, developers often identify the precise moment in execution where the actual behavior diverges from the intended one. However, rectifying this discrepancy poses a challenge. This paper introduces TraceFixer, a novel method designed to anticipate code edits that align the program's behavior with the desired outcomes. TraceFixer utilizes a neural program repair model trained not only on source code edits but also on snippets of runtime traces. The model takes as input a partial execution trace of the erroneous code, automatically obtained through code instrumentation, along with the desired program state at the point of divergence, provided by the user, typically via an interactive debugger. Unlike existing program repair techniques, which aim for similar goals but overlook execution traces and desired program states, TraceFixer fundamentally integrates this information into its repair process. Evaluation on single-line Python errors demonstrates that incorporating execution traces enhances bug-fixing accuracy by 13% to 20% compared to baseline techniques that solely learn from source code edits. Application of TraceFixer to real-world Python code further validates its effectiveness, successfully resolving 10 out of 20 instances [3].

Ensuring smooth handling of stop and start events is crucial for Android apps to maintain state information without loss during transitions. However, developers frequently overlook the implementation of necessary logic for saving and restoring app states, leading to data loss issues upon events such as moving the app to the background or rotating the screen. These shortcomings can result in usability issues and unexpected crashes, significantly impacting user experience. To facilitate research and experimentation in addressing these challenges, this paper introduces a public benchmark comprising 110 data loss faults observed in Android apps. The benchmark, available on GitLab, includes both faulty and fixed versions of apps (where applicable), along with test cases for automatic reproduction of the problems and supplementary information to assist researchers in their investigations [19].

Resource leaks, where programs fail to release previously acquired resources, present a prevalent issue in Android applications. Despite the availability of existing techniques for automatic leak detection, writing leak-free programs remains challenging. This challenge is partly attributed to Android's event-driven programming model, which complicates the understanding of the application's control flow. This paper introduces PlumbDroid, a technique for automated detection and resolution of resource leaks in Android apps. PlumbDroid constructs a concise abstraction of an app's control flow to identify execution traces potentially leading to leaks. Leveraging this information, it automatically generates fixes by adding release operations at appropriate locations to rectify the leaks without affecting resource usage elsewhere in the application. Empirical evaluation on resource leaks from the DroidLeaks curated collection demonstrates PlumbDroid's scalability, precision, and ability to produce correct fixes for various resource leak bugs. PlumbDroid successfully detected and repaired 50 leaks affecting 9 commonly used Android resources, including those collected by Droi-

dLeaks, with an average detection and repair time of just 2 minutes per leak. Moreover, PlumbDroid exhibits favorable comparison against Relda2/RelFix, the only other fully automated approach for Android resource leak repair, by detecting more leaks with higher precision and generating smaller fixes. These findings underscore PlumbDroid's potential to significantly enhance the quality of Android applications in practice [2].

The Android platform presents a unique framework for app development, where non-compliance can result in severe bugs. With the platform's rapid evolution, developers heavily rely on its APIs, necessitating frequent app updates to avoid compatibility issues. Additionally, Android apps encounter various device-specific and memory-related issues due to deployment on a wide range of memory-constrained devices. Addressing these Android-specific challenges is crucial during app development and maintenance to prevent serious crashes. In this paper, an empirical study is conducted to investigate and characterize various Android-specific crash bugs, including their prevalence, root causes, and solutions. Through the analysis of 1,862 confirmed crash reports from 418 open-source Android apps, insights are provided to aid developers in understanding, preventing, and fixing Android-specific crash bugs. Furthermore, these findings can inform the design of effective bug detection tools for Android apps, benefiting both developers and researchers [9].

Automated Program Repair (APR) has garnered significant attention across various programming languages and platforms. Despite the existence of numerous APR techniques and benchmarks, there remains a gap in leveraging APR methods for mobile development, particularly within the Android ecosystem. To address this gap, DroidBugs, an introductory benchmark specifically tailored for Android projects, is introduced. DroidBugs is derived from the analysis of 360 open-source Android projects, each with a substantial user base of more than 5,000 downloads. This benchmark comprises 13 single-bugs categorized by the type of test that exposed them, providing a diverse set of challenges for APR evaluation. Through the utilization of the APR tool Astor4Android and two common fault localization strategies, the difficulty of identifying and rectifying these mobile bugs is assessed. DroidBugs stands as a valuable resource for advancing research in automated program repair within the mobile development domain [1].

Mutation testing serves as a crucial tool for evaluating the effectiveness of test suites and guiding test case generation or prioritization. While mutants generally represent real faults, effective mutation testing in specific application domains, such as Android apps, requires domain-specific mutation operators. This paper presents MDroid+, a framework designed for effective mutation testing of Android apps. The framework begins by systematically devising a taxonomy comprising 262 types of Android faults, categorized into 14 groups, through manual analysis of 2,023 software artifacts from various sources like bug reports and commits. Subsequently, a set of 38 mutation operators is identified and implemented, with 35 of them integrated into an infrastructure for

automatically seeding mutations in Android apps. The taxonomy and proposed operators are evaluated based on stillborn/trivial mutants generated and their capacity to represent real faults in Android apps, comparing them with other established mutation tools [14].

API misuses present significant challenges, often resulting in software crashes and security vulnerabilities. Detecting and repairing such misuses is particularly challenging as correct API usage may be unclear to developers of client programs. This paper presents the first empirical study to assess the capabilities of existing automated bug repair tools in repairing API misuses, a previously unexplored class of bugs. The study evaluates and compares 14 Java test-suite-based repair tools (11 proposed before 2018 and three afterwards) using a manually curated benchmark, APIREPBENCH, comprising 101 API misuses. An extensible execution framework, APIARTY, is introduced to automatically execute multiple repair tools. Results indicate that the repair tools successfully generate patches for 28% of the considered API misuses. While earlier tools demonstrate efficiency with a median execution time of 3.87 minutes and a mean execution time of 30.79 minutes, more recent tools exhibit lower efficiency, being 98% slower. The generated patches primarily address API misuses in categories such as missing null check, missing value, missing exception, and missing call. Although most patches are plausible (65%), only a few are semantically correct compared to human-generated patches (25%). Findings suggest that future repair tools should focus on localizing complex bugs, handling timeout issues, and configuring large software projects. Both APIREPBENCH and APIARTY are publicly available for researchers to evaluate repair tools' capabilities in detecting and fixing API misuses [13].

Automated Program Repair (APR) aims to enhance software reliability by generating patches for flawed programs. While Code Language Models (CLMs) have demonstrated effectiveness in various software tasks such as code completion, their capabilities in bug fixing have not been comprehensively evaluated or fine-tuned for the APR task. This study pioneers in evaluating ten CLMs across four APR benchmarks, revealing that the top-performing CLM, in its original state, rectifies 72% more bugs compared to state-of-the-art deep learning (DL)-based APR techniques. Additionally, one APR benchmark introduced in this study addresses data leakage concerns, ensuring fair evaluation. Furthermore, this work pioneers in fine-tuning CLMs with APR training data, demonstrating significant improvements ranging from 31% to 1,267% over existing DL-based APR techniques, enabling them to fix 46% to 164% more bugs. Exploring the impact of buggy lines, the study indicates that unaltered CLMs struggle to leverage such lines effectively, while fine-tuned CLMs may overly rely on them. Lastly, the study assesses the size, time, and memory efficiency of various CLMs, shedding light on their performance metrics. This research not only unveils promising directions for the APR domain, particularly in fine-tuning CLMs with APR-specific designs, but also underscores the importance of fair and comprehensive CLM evaluations, advocating for transparent reporting of open-source repositories

used in pre-training data to address data leakage concerns [10].

Unit tests are vital for enhancing software quality, yet crafting effective tests from scratch can be daunting. Recommending existing tests for semantically similar functions can alleviate this burden, but manually modifying these recommended tests remains challenging. Various code elements in the recommended tests need to be comprehended by developers to replace them accurately with semantically similar elements from the target application. To address this challenge, a test migration or reuse technique is proposed to automatically transform code elements in recommended tests and migrate them to the target application. The paper first identifies the types of code transformations needed for successful test migration. External participants are recruited to create JTESTMIGBENCH, a benchmark comprising 510 manually migrated JUnit tests for 186 methods from five popular libraries. The paper then analyzes the code changes in the migrated tests to develop JTESTMIGTAX, a taxonomy of test code transformation patterns. The contributions lay the groundwork for developing automated unit test migration or reuse techniques [8].

Automated Program Repair (APR) techniques aim to enhance software reliability by automatically generating patches for buggy programs. Recent APR methods utilizing deep learning (DL) encounter challenges due to syntactically or semantically incorrect patches within the patch space. These erroneous patches often deviate from the syntactic and semantic norms of source code, rendering them ineffective in fixing bugs. To address this issue, KNOD, a DL-based APR approach, is proposed. KNOD integrates domain knowledge to guide patch generation directly and comprehensively, introducing two significant innovations: (1) a novel three-stage tree decoder that generates Abstract Syntax Trees of patched code based on the inherent tree structure, and (2) a novel domain-rule distillation method that incorporates syntactic and semantic rules and teacher-student distributions to inject domain knowledge into the decoding process during both training and inference phases. KNOD is evaluated on three widely-used benchmarks, successfully fixing bugs in Defects4J v1.2, QuixBugs, and additional Defects4J v2.0 benchmarks, outperforming all existing APR tools [11].

Numerous automated test generation techniques have been devised to assist developers in writing tests, primarily focusing on increasing coverage or generating exploratory inputs for full automation. However, existing techniques often fall short in achieving more semantic objectives, such as generating tests to replicate a given bug report. Reproducing bugs is crucial, as approximately 28% of tests added in open-source repositories are due to issues. Existing failure reproduction techniques predominantly handle program crashes, neglecting a broader spectrum of bug reports. To address this gap, LIBRO, a framework utilizing Large Language Models (LLMs) capable of executing code-related tasks, is proposed. Post-processing steps are emphasized to ascertain LLMs' effectiveness and rank generated tests based on validity. Evaluation of LIBRO on the Defects4J benchmark demonstrates its capability to generate failure-reproducing

test cases, with a bug-reproducing test ranked first for 149 bugs. Additionally, when evaluated against bug reports submitted after the termination of LLM training data collection, LIBRO produces bug-reproducing tests for 32% of the studied bug reports, indicating its potential to significantly enhance developer efficiency by automating test generation from bug reports [12].

4 Methodology

4.1 Research Design

To conduct a systematic review of the existing literature on automated vulnerability repair in Android applications, we identified the research gap and formulated the research question. The research diagram is shown in 4.1.

4.2 Data Collection

The second step of the methodology design is to collect a dataset of 80 vulnerable code snippets sourced from Google Android Security Bulletins, which provide official information and patches (see image 4.2) for security vulnerabilities for Android. The purpose of this step is to obtain a representative and authentic sample of real-world Android vulnerabilities and their corresponding fixes, which will be used to evaluate the performance of the LLM-based tools.'

The data collection will be performed using a Python script that will scrape the data from the web and store it into MongoDB, a NoSQL database. The script will follow these steps:

Access the Google Android Security Bulletin website: The script will access the website Android Security Bulletin, which contains the monthly security bulletins for Android devices, dating back to 2015. The script will parse the HTML content of the website and extract the links to the individual bulletins for each month and year.

Retrieve the bulletins for the selected period: The script will iterate over the links to the bulletins and retrieve the bulletins for the selected period, which is from January 2020 to December 2020. The script will parse the HTML content of each bulletin and extract the information about the security vulnerabilities, such as the CVE number, the severity level, the affected versions, the description,

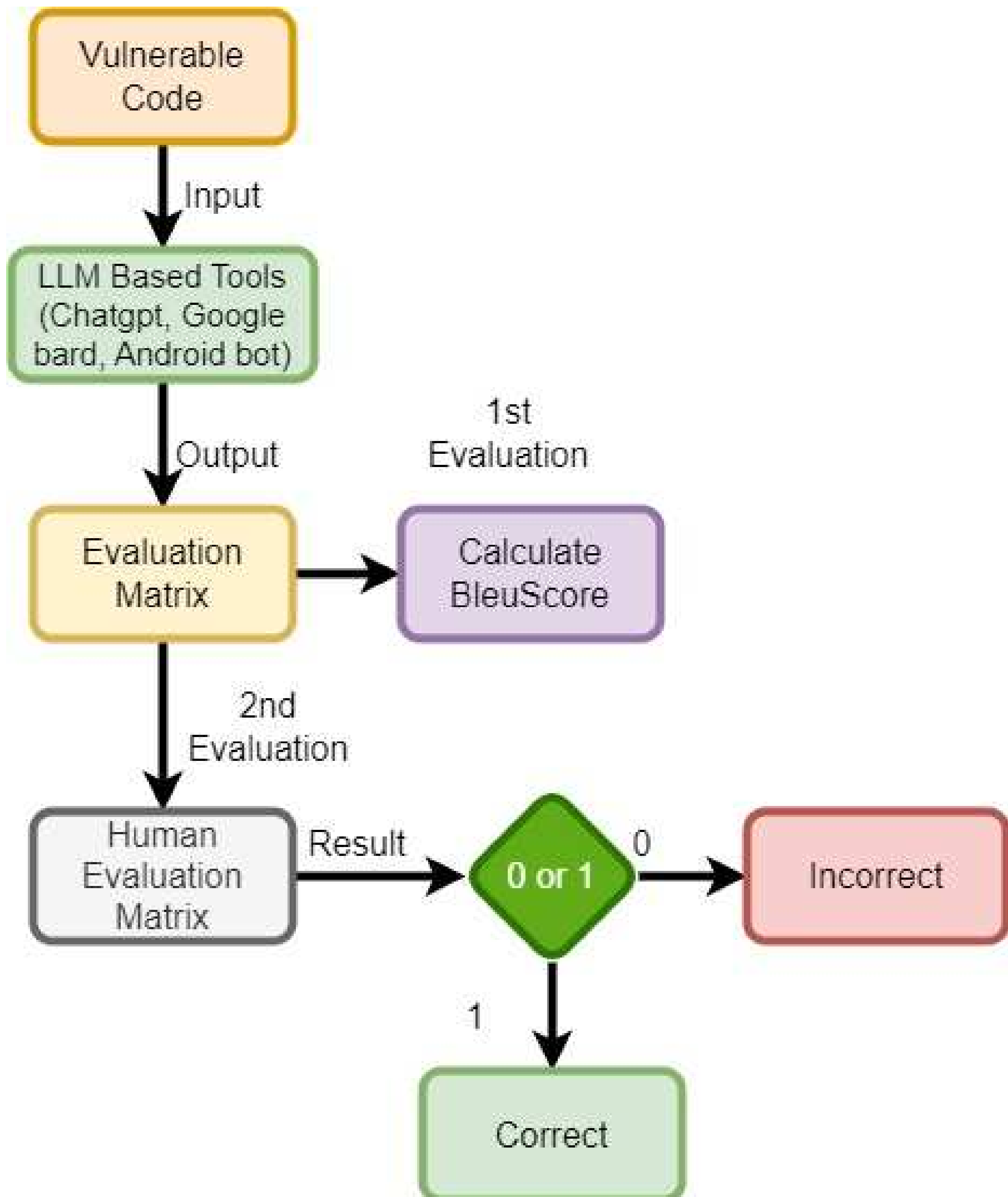

Figure 4.1: Research Diagram

android / platform / frameworks / base / 9bdd9d274ac4ce77c0e8d649141ceea115b1ddbe

```
   commit  9bdd9d274ac4ce77c0e8d649141ceea115b1ddbe        [log] [tgz]
   author  William Loh <wloh@google.com>                   Fri Aug 05 20:25:27 2022 +0000
committer  William Loh <wloh@google.com>                   Tue Sep 06 18:45:38 2022 +0000
     tree  16ee84e4622fe868ee68e225a0fe2ac0f8730439
   parent  5421a3f422b796bf3b513320d484fbd33486f365 [diff]
```

```
Limit length and number of MIME types you can set

Limit character length of MIME types to 255. If this length is exceeded
then a IllegalArugmentException is thrown. The number of MIME types that
can be set is also limited to 500 per MIME group with the number of
total MIME Groups also limited to 500. A IllegalStateException is thrown if this number is exceeded.

Bug: 237291548
Test: Installed and ran POC app from b/237291548
Change-Id: I1d57e674f778cfacdc89225ac3273c432a39af63
Merged-In: I1d57e674f778cfacdc89225ac3273c432a39af63
```

Bug Message

services/core/java/com/android/server/pm/PackageManagerService.java [diff]
services/core/java/com/android/server/pm/pkg/parsing/ParsingPackageImpl.java [diff]

Buggy and Correct Code

Figure 4.2: Data Available for a Specific Vulnerability

and the patch link.

Filter the vulnerabilities by severity and version: The script will filter the vulnerabilities by their severity and version, and select only those that have a high or critical severity level, and that affect Android 10 or 11, which are the latest versions of Android as of 2020. The script will also exclude any vulnerabilities that do not have a patch link, or that have a patch link that is not accessible or valid.

Download the patch files and extract the code snippets: The script will download the patch files for the selected vulnerabilities, which are in the form of diff files that show the changes made to the source code to fix the vulnerability For reference, please see Figure 4.3. The script will parse the diff files and extract the code snippets that correspond to the vulnerable and fixed versions of the code. The script will also ensure that the code snippets are valid and complete Java or Kotlin code, and that they are not too long or too short for the evaluation purpose.

Store the code snippets and the metadata into MongoDB: The script will store the code snippets and the metadata into MongoDB, a NoSQL database that allows flexible and scalable storage of data. The script will create a collection called vulnerabilities in MongoDB, and store each vulnerability as a document with the following fields: CVE number, severity level, affected version, description, patch link, vulnerable code snippets, see on table 4.1.

Figure 4.3: Format of Availability of Buggy Code and Correct Code for a Specific Vulnerability

Parameter	Description
id	unique id which identifies the corresponding vulnerability
CVE	unique CVE code representing the vulnerability
link	Link to the corresponding CVE for complete data
commit	Unique commit ID providing commit details
author	Name and email of the author committing the solution
committer	Name and email of the committer committing the solution
type	Type of vulnerability
Severity	Indicates severity, either "warning" or "critical"
Message	Vulnerability message
Changes	Stores updated and vulnerable code

Table 4.1: Dataset parameters

The expected outcome of the data collection is to obtain a dataset of 80 vulnerable code snippets and their corresponding fixes, sourced from Google Android Security Bulletins, which will be used to evaluate the performance of the LLM-based tools. The dataset will be representative and authentic, as it will reflect the real-world Android vulnerabilities and their official patches. The dataset will also be stored in a flexible and scalable database, which will facilitate data retrieval and manipulation for evaluation purposes.

The final dataset consists of 80 Android security vulnerabilities code. It contains three columns:

1. an id that is unique and identifies the vulnerability.
2. the language of the snippet.
3. the vulnerability and its context.

4. the correct code

Additionally, to further enrich this dataset, we download all the "diff" files representing both the erroneous (wrong) and corrected (right) code. These "diff" files allow for a detailed comparison, enabling the differentiation between the flawed code and its rectified version. Through this comprehensive approach, researchers and developers gain access to a multifaceted resource, facilitating in-depth exploration and mitigation strategies for Android security vulnerabilities.

In addition to tracking different CVEs from which vulnerabilities are extracted, these data are stored in a MongoDB database in the form of a document. This document encapsulates various parameters pertinent to the vulnerabilities being cataloged. The schema for the MongoDB document, illustrated in Figure 4.4, encompasses the following key elements:

CVE ID (CVE-XXXX-XXXX): This field denotes the Common Vulnerabilities and Exposures (CVE) identifier associated with the vulnerability. It serves as a unique reference point for the vulnerability.

Description: A textual description outlining the nature and characteristics of the vulnerability. This description provides insights into the specific security issue and its potential impact.

Published Date: Indicates the date when the vulnerability information was made publicly available. This timestamp aids in understanding the timeline of the vulnerability disclosure and its relevance to mitigation efforts.

CVSS Score (Common Vulnerability Scoring System): The CVSS score quantifies the severity of the vulnerability on a scale from 0 to 10, with higher scores indicating greater severity. This metric assists in prioritizing remediation efforts based on the criticality of the vulnerability.

Affected Systems: Specifies the systems, software, or components susceptible to exploitation due to the vulnerability. Understanding the affected assets aids in devising targeted mitigation strategies.

References: Includes links or references to external resources such as advisories, patches, or research articles related to the vulnerability. These references serve as additional sources of information for understanding and mitigating the security issue.

Comments/Notes: Allows for the inclusion of supplementary comments or notes relevant to the vulnerability. This section may contain additional context, mitigation recommendations, or internal annotations.

```
"_id": {
  "$oid": id
},
"CVE": cve,
"References": [
  "text": "text",
  "link": "https://android.googlesource.com/kernel/common/+/201d5f4a3ec1",
  "metaData": {
    "commit": "201d5f4a3ec12c639ecf2284da45a1ebd9e2141d",
    "author": "Carlos Llamas <cmllamas@google.com>",
    "committer": "Carlos Llamas <cmllamas@google.com>",
    "tree": "4f8877b336f1a73b2869c8e7b1910ed153d47877",
    "parent": "20af947ec911d9b4f59745115199d6f011909186 [diff]"
  },
  "metadataMessage": message,
  "changes": [
    {
      "diff": {
        "description":
        "files": [ file1,file2
        ]
      },
      "diffUnified": changes
    }]],
"Type": "EoP",
"Severity": "High",
"Subcomponent": "Binder driver"
```

Figure 4.4: Parameters stored in the database of Vulnerabilities

Data is available here : https://github.com/SaadAhmed1122/Android-Security-Bulletin-cve/tree/main

4.3 Tool Selection

The third step of the methodology design is to select three LLM-based tools for automated Android security vulnerability repair: ChatGPT, Google Bard, and Android Studio Bot. The purpose of this step is to choose the tools that will be evaluated and compared in terms of their effectiveness and reliability in generating suggestions for fixing vulnerable code snippets.

The tool selection will be based on the following criteria:

Availability and accessibility: The tools should be publicly available and accessible, either as open-source software, web applications, or APIs. The tools should also have clear and comprehensive documentation and instructions for using them.

Relevance and applicability: The tools should be relevant and applicable for the task of automated vulnerability repair in Android applications, and should be able to handle Java or Kotlin code snippets. The tools should also be able to generate suggestions based on the description or the severity of the vulnerability, as provided by the Google Android Security Bulletins.

Popularity and novelty: The tools should be popular and novel, meaning that they should have a high number of users, citations, or media coverage, and that they should represent the state-of-the-art of LLM-based tools for automated vulnerability repair.

Based on these criteria, the following three tools are selected for the evaluation:

ChatGPT: ChatGPT is an LLM-based tool that uses a chatbot interface to generate suggestions for fixing code snippets, based on natural language prompts. ChatGPT is based on GPT-3, a large-scale LLM that can generate natural-sounding text or code snippets based on a given context or prompt. ChatGPT is available as a web application at https://chatgpt.com/, and can handle various programming languages, including Java and Kotlin. ChatGPT can generate suggestions for fixing code snippets based on the description or the severity of the vulnerability, by using natural language prompts such as fix this code snippet to prevent SQL injection or make this code snippet more secure.

Google Bard: Google Bard is an LLM-based tool that uses a web interface to generate suggestions for fixing code snippets, based on the severity of the vulnerability. Google Bard is based on BART, a large-scale LLM that can perform various text generation tasks, such as text summarization, translation, classification, and completion. Google Bard is available as a web application at https://ai.google/research/bard, and can handle various programming languages, including Java and Kotlin. Google Bard can generate suggestions for fixing code snippets based on the severity of the vulnerability, by using a slider that ranges from low to high, and that indicates the level of security improvement required for the code snippet.

Android Studio Bot: Android Studio Bot is an LLM-based tool that uses an IDE plugin to generate suggestions for fixing code snippets, based on the

description of the vulnerability. Android Studio Bot is based on CodeBERT, a large-scale LLM that can perform various code-related tasks, such as code summarization, translation, classification, and completion. Android Studio Bot is available as an open-source plugin for Android Studio, an IDE for Android application development, at Android Studio Bot. Android Studio Bot can generate suggestions for fixing code snippets based on the description of the vulnerability, by using a text box that allows the user to enter the description, and that triggers the suggestion generation.

The expected outcome of the tool selection is to choose three LLM-based tools for automated Android security vulnerability repair: ChatGPT, Google Bard, and Android Studio Bot. These tools are publicly available and accessible, relevant and applicable, and popular and novel, and they represent the state-of-the-art of LLM-based tools for automated vulnerability repair. These tools will be evaluated and compared in terms of their effectiveness and reliability in generating suggestions for fixing vulnerable code snippets.

4.4 Prompt

In order to fully utilize Language Model (LLM) modelslike GPT (Generative Pre-trained Transformer), Google Bard and Android Studio Bot modelsfor a variety of tasks, from text production to language interpretation, prompts are essential. These cues or instructions serve as a means of directing the model's creation or comprehension process. Prompts can be categorized based on their intended use and the type of response they are designed to elicit from a Large Language Model (LLM). Here are some common categories:

- Classification Prompts: Used when you need to classify text into specific categories[15].
- Closed-ended Prompts: Ideal for obtaining specific information or a yes/no answer[6].
- Contextual Prompts: Provide additional context to guide the LLM's response[7].
- Counterfactual Prompts: Involve hypothetical scenarios that differ from known facts[23].
- Exploratory Prompts: Aim to explore a topic or generate ideas without specific constraints[4].
- Generative Prompts: Designed to generate creative or original content.
- Hypothetical Prompts: Pose hypothetical questions or scenarios to explore possible outcomes.
- Interactive Prompts: Engage the LLM in a dialogue or interactive session.

These categories help in structuring prompts to achieve more accurate and relevant responses from LLM Models.

4.4.1 Prompt Used

In order to obtain results from the LLM model, we utilize the following prompt template for generating fixes, and we employ the same template for all other LLM models.

The template employed to formulate the specific prompt is as follows:

1. Following is a vulnerable code. Correct it.
2. Message of the vulnerable code which is given in the commit.
3. Buggy Code

In Image 4.5, the example illustrates a specific vulnerability within an input prompt.

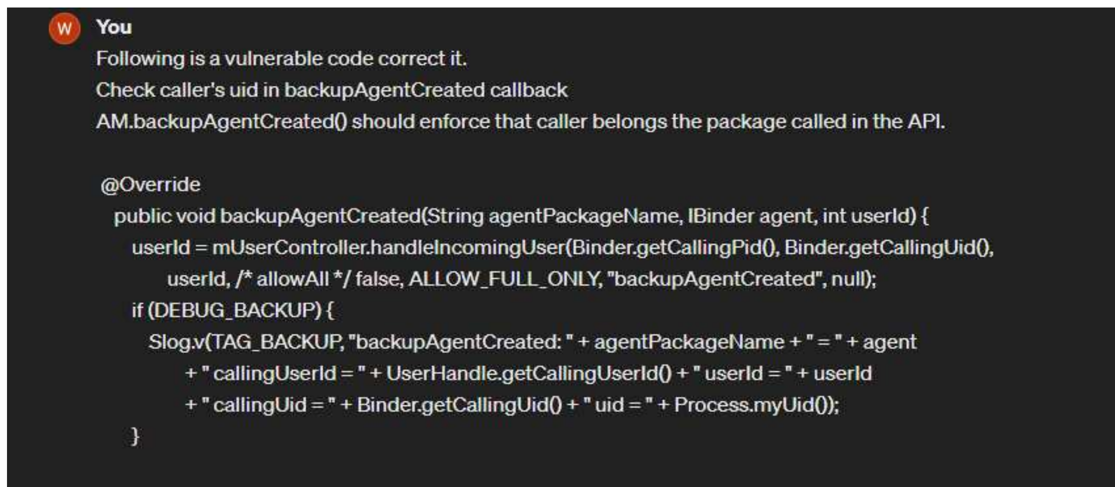

Figure 4.5: Example of Prompt Used to generate result from LLM Models

4.5 Pre-Processing Phase

The preprocessing phase is a crucial step that involves cleaning and preparing the data before processing. In our thesis, we conducted preprocessing after obtaining the results from the LLM models and processing the results to prepare them for evaluation using a matrix.

4.5.1 Pre-processing of LLM Model outputs

In the results of LLM models, we receive additional data alongside the respective code. We disregard this supplementary information, such as code descriptions and examples, and solely extract the corresponding code for further analysis. This uniform approach is applied consistently across all other LLM models. In Figure 4.6, despite the abundance of accompanying information and code in the results, we focus solely on the code for evaluation purposes.

4.6 Evaluation Metrics

In assessing the performance of Language Model (LLM)-based tools for automated Android security vulnerability repair, several evaluation metrics will be employed to gauge their effectiveness and reliability in generating fixes for vulnerable code snippets. These metrics encompass both qualitative and quantitative measures to provide a comprehensive evaluation:

BLEU Score: The Bilingual Evaluation Understudy (BLEU) score is a metric commonly used to evaluate the quality of machine-generated text, such as code snippets generated by LLM-based tools. BLEU compares the similarity between the generated code snippet and the reference (i.e., the actual fix provided by Google or other authoritative sources). A higher BLEU score indicates a closer

To address the vulnerability in the provided code, where the token is stripped from the `mAllowlistToken` but not from `notification.publicVersion`, you should ensure that any sensitive tokens or data are properly sanitized across all relevant components.

Here's the corrected code where the token is stripped from both `mAllowlistToken` and `notification.publicVersion`:

java Copy code

```java
public void setAllowlistToken(@Nullable IBinder token) {
    mAllowlistToken = token;
    if (notification != null && notification.publicVersion != null) {
        notification.publicVersion.setAllowlistToken(null);
    }
}
```

In this corrected version, when setting the allowlist token for the main object, it also checks if the `publicVersion` exists and if so, it sets its allowlist token to null as well, ensuring that both objects are properly sanitized.

Figure 4.6: Proposed fix for Android specific vulnerability by ChatGPT.

match between the generated fix and the reference, reflecting greater syntactic and semantic correctness.

Human Evaluation: Human evaluation involves manual assessment by human evaluators, who will review the generated fixes and provide subjective ratings based on criteria such as readability, maintainability, and relevance to the vulnerability description. This qualitative assessment complements quantitative metrics by capturing aspects of code quality that may not be captured by automated measures.

4.7 Experimental Setup

The experimental setup involves conducting experiments to evaluate the performance of the selected LLM-based tools for automated Android security vulnerability repair. The following components constitute the experimental setup:

Dataset Preparation: The dataset of vulnerable code snippets, sourced from Google Android Security Bulletins, will be prepared by selecting a representative sample of vulnerabilities affecting Android 10 or 11 and categorized by severity level. Each code snippet will be paired with its corresponding fix provided by Google for benchmarking purposes.

Tool Configuration: The selected LLM-based toolsChatGPT, Google Bard,

and Android Studio Botwill be configured and set up for generating fixes based on the given code snippets and vulnerability descriptions. Any required preprocessing steps, such as input formatting or model fine-tuning, will be performed to optimize tool performance.

Experiment Execution: The experiments will involve feeding each vulnerable code snippet and its corresponding vulnerability description into the LLM-based tools to generate candidate fixes. The generated fixes will be evaluated using the established evaluation metrics, including BLEU score and human evaluation ratings.

4.8 Data Analysis Techniques

The analysis of experimental results will entail applying various data analysis techniques to interpret and derive insights from the evaluation outcomes. Key data analysis techniques include:

Statistical Analysis: Quantitative metrics with BLEU score, analyzed statistically to compare the performance of different LLM-based tools.

Qualitative Analysis: Human evaluation ratings and qualitative feedback will undergo thematic analysis to identify common themes and patterns in evaluators' perceptions of the generated fixes. Qualitative insights will complement quantitative metrics by providing nuanced interpretations of tool performance and identifying areas for improvement.

Visualization: Visualization techniques, such as box plots, histograms, and heat-maps, may be utilized to present the distribution and relationships between evaluation metrics. Visual representations facilitate the interpretation of complex data and aid in communicating findings to stakeholders effectively.

By employing a combination of evaluation metrics, experimental setup, and data analysis techniques, the methodology aims to provide a rigorous and comprehensive evaluation of LLM-based tools for automated Android security vulnerability repair.

5 Results

5.1 Results Overview

In this section, we present real-world examples of Android security vulnerabilities and demonstrate the application of Language Model (LLM)-based tools in vulnerability repair. We assess the effectiveness and efficiency of LLM-based repair strategies through comprehensive case studies conducted on a diverse set of Android applications.

Our analysis includes a detailed examination of various vulnerability types, such as input validation flaws, authentication bypass vulnerabilities, and insecure data storage issues. By applying LLM-based techniques, we successfully identify and mitigate these vulnerabilities, showcasing the potential of automated repair solutions in enhancing Android application security.

Through our case studies, we evaluate the performance of LLM-based tools in terms of repair accuracy, speed, and scalability. We compare the results obtained from LLM-based repairs with traditional manual approaches to highlight the advantages and limitations of each method. Additionally, we discuss the challenges encountered during the repair process and propose strategies for improving the effectiveness of LLM-based vulnerability repair in real-world scenarios.

Overall, our findings underscore the importance of incorporating LLM-based tools into the Android security development lifecycle. By leveraging the capabilities of these advanced language models, developers can efficiently identify and address security vulnerabilities, thereby enhancing the overall resilience of Android applications against potential cyber threats.

5.2 Evaluation Matrix

An evaluation matrix, also known as a decision matrix or criteria matrix, is a systematic tool used to assess and compare different options or alternatives based on a set of predetermined criteria. It serves as a structured approach to decision-making, especially in complex situations where multiple factors need to be considered.

In the context of this thesis, we employ the CodeBLEU score evaluation matrix to quantitatively evaluate the performance of our proposed vulnerability repair techniques. The CodeBLEU score, inspired by the BLEU (Bilingual Evaluation Understudy) metric commonly used in natural language processing tasks, provides a standardized measure of similarity between the original vulnerable code and the repaired code. By analyzing various aspects such as n-gram matches, syntactic similarities, and dataflow consistency, the CodeBLEU score offers valuable insights into the effectiveness of our repair strategies.

Additionally, we incorporate human evaluation as part of our assessment process to complement the automated metrics provided by the CodeBLEU score. Human evaluators, typically experienced software developers or security experts, review the repaired code samples and provide qualitative feedback on factors such as readability, maintainability, and overall security improvements. This human-centric approach enables us to capture nuanced aspects of vulnerability repair that may not be fully captured by automated metrics alone.

By combining both automated evaluation metrics and human judgment, we aim to comprehensively assess the performance of our vulnerability repair techniques. This hybrid evaluation approach not only provides objective measures of effectiveness but also ensures that the repaired code meets the practical requirements and expectations of real-world developers and security practitioners.

5.2.1 Code Bleu

We use the CodeBleu Evaluation Matrix to compare the results of correct code and predicted code. BLEU is calculated using the standard BLEU metric, where BLEU weight represents the weighted n-gram match. This is obtained by comparing the hypothesis code and the reference code tokens with varying weights. MatchAST represents the syntactic AST match, delving into the syntactic information of the code, while MatchDF denotes the semantic dataflow match, considering the semantic similarity between the hypothesis and the reference. The weighted n-gram match and the syntactic AST match gauge grammatical correctness, whereas the semantic data-flow match assesses logical correctness.

The evaluation metrics include:

N-gram Match: This metric measures the similarity between n-grams (sequences of n items, such as words or tokens) in the predicted code and the reference code.

Weighted N-gram Match: Similar to N-gram Match, but with weighted n-grams, where different weights are assigned to different n-gram matches based on their importance. (See Figure 5.1)

Dataflow Match: Dataflow Match evaluates the semantic similarity between the predicted and reference code in terms of data flow analysis. (See Figure 5.3)

Syntactic AST Match: Syntactic AST Match analyzes the similarity between machine and human translations in terms of their Abstract Syntax Trees (ASTs). An AST represents the hierarchical structure of the program or sentence, capturing its syntactic components and their relationships. (See Figure 5.2)

CodeBleu Score: The CodeBLEU score combines lexical, syntactic, and semantic aspects to quantify the similarity between the predicted and reference code, with higher scores indicating better code repair quality.

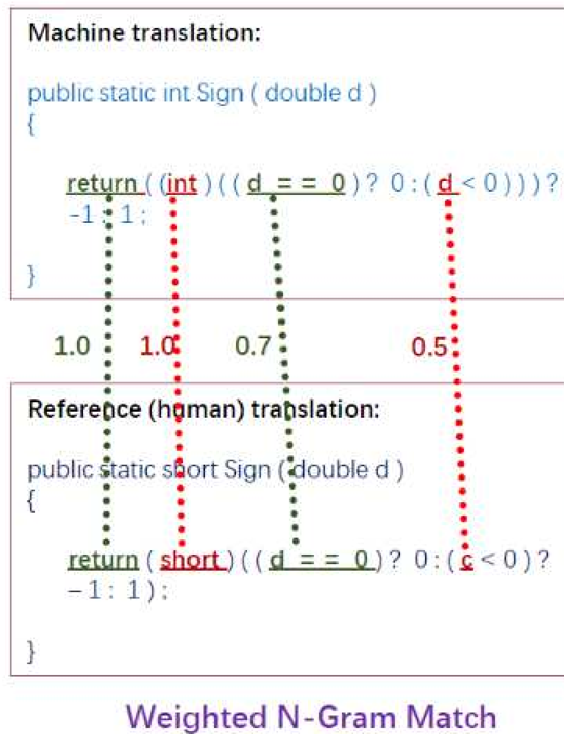

Figure 5.1: CodeBlue weighted N-Gram match [17].

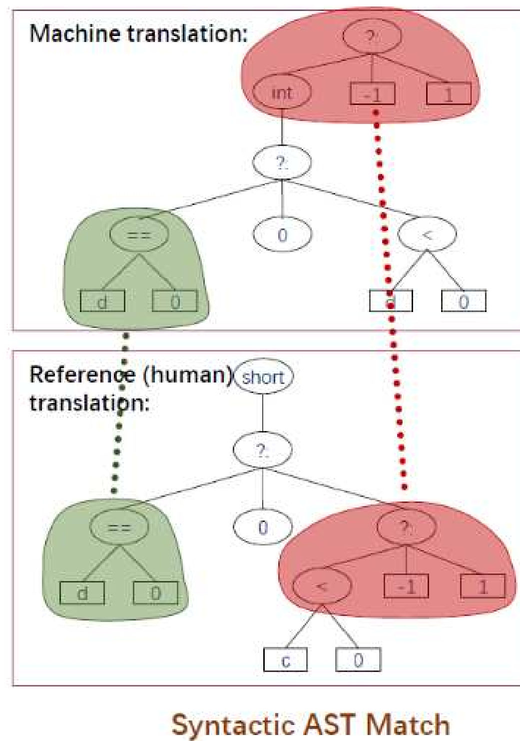

Figure 5.2: CodeBlue syntactic AST match [17]

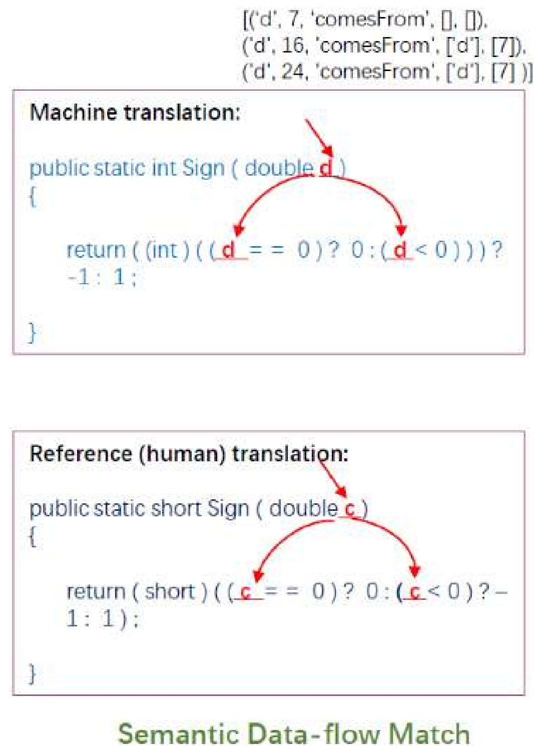

Figure 5.3: CodeBlue semantic data flow [17]

The equation 5.2.1 represents a scoring mechanism called CodeBLEU, which is used to evaluate the similarity or quality of generated code compared to reference code. Each component of the equation corresponds to a different aspect of similarity or match between the generated code and the reference code.

$$\begin{aligned} \text{CodeBLEU} = \alpha \times n_{\text{gram_match}} \\ + \beta \times \text{weighted_n_gram_match} \\ + \gamma \times \text{syntactic_match} \\ + \delta \times \text{dataflow_match} \end{aligned} \tag{5.2.1}$$

- α represents the weight given to the match of n-grams in the generated and reference code.
- β represents the weight given to the match of weighted n-grams in the generated and reference code.
- γ represents the weight given to the match of syntactic structures between the generated and reference code.
- δ represents the weight given to the match of dataflow patterns between the generated and reference code.

5.2.2 Results Table

The following tables show the clear difference between those LLM Models and represent the clear score which is calculated by CodeBlue.

Code Blue Results of ChatGPT-3.5

S:No	N-gram Match	Weighted N-gram Match	Syntactic Match	Dataflow Match	CodeBlue Score
1	0.324	0.306	0.684	1	0.578
2	0.395	0.395	0.583	0.833	0.551
3	0.634	0.607	0.667	1	0.727
4	0.634	0.607	0.667	1	0.877
5	0.853	0.829	0.826	1	0.727
...	...	...	...	...	...
80	0.612	0.769	0.667	1	0.762

Table 5.1: Evaluation Metrics for evaluating CodeBLEU of ChatGPT-3.5 for Android Java code vulnerabilities

This table 5.1 displays the evaluation metrics used to assess the CodeBLEU score of ChatGPT-3.5 in the context of identifying vulnerabilities in Android Java code. Each row in the table corresponds to a specific evaluation instance, labeled by S:No (Serial Number).

Additionally, the total Code Blue score of ChatGPT-3.5 across all evaluation instances is provided as 0.490674941334816.

This table and the codeblue score provide insights into the performance of ChatGPT-3.5 in repairing Android Java code vulnerabilities, offering a quantitative assessment of its effectiveness.

CodeBlue Result of Google Bard

S:No	N-gram Match	Weighted N-gram Match	Syntactic Match	Dataflow Match	CodeBlue Score
1	0.171	0.277	0.473	0.70	0.405
2	0.142	0.233	0.333	0.50	0.302
3	0.013	0.026	0.333	0.50	0.218
4	0.075	0.124	0.666	1.00	0.466
5	0.053	0.084	0.652	0.85	0.411
...	...	...	...	...	...
80	0.802	0.811	0.727	1.00	0.835

Table 5.2: Evaluation Metrics for evaluating CodeBLEU of Google Bard for Android Java code vulnerabilities

This table 5.2 presents the evaluation metrics used to assess the CodeBLEU score of Google Bard in the context of identifying vulnerabilities in Android Java code. Each row in the table corresponds to a specific evaluation instance, labeled by S:No (Serial Number).

Additionally, the total Code Blue score of Google Bard across all evaluation instances is provided as 0.43066895636352526.

This table and the codeblue score offer insights into the performance of Google Bard in repairing Android Java code vulnerabilities, providing a quantitative assessment of its effectiveness.

CodeBlue Result of Android Studio Bot

S:No	N-gram Match	Weighted N-gram Match	Syntactic Match	Dataflow Match	CodeBlue Score
1	0.504	0.581	0.736	0.80	0.655
2	0.020	0.026	0.583	0.50	0.282
3	0.043	0.044	0.416	1.00	0.376
4	0.077	0.083	0.652	0.28	0.274
5	0.043	0.044	0.416	1.00	0.376
...	...	...	...	...	...
80	0.874	0.929	0.757	1.00	0.890

Table 5.3: Evaluation Metrics for evaluating CodeBLEU of Android Studio Bot for Android Java code vulnerabilities

This table 5.3 displays the evaluation metrics used to assess the CodeBLEU score of Android Studio Bot in the context of identifying vulnerabilities in Android Java code. Each row in the table corresponds to a specific evaluation instance, labeled by S:No (Serial Number).

Additionally, the total Code Blue score of Android Studio Bot across all evaluation instances is provided as 0.5297597933431579.

This table and the codeblue score offer insights into the performance of Android Studio Bot in repairing Android Java code vulnerabilities, providing a quantitative assessment of its effectiveness.

OVERALL CODEBLUE SCORE OF THESE THREE LLMS

	Chatgpt-3.5	Google Bard	Android Studio Bot
Overall CodeBlue Score	0.49	0.43	0.52

Table 5.4: Average of overall result

Table 5.4 presents the overall CodeBlue scores of the LLMs (Language Model-based tools) used for vulnerability repair, including ChatGPT-3.5, Google Bard, and Android Studio Bot.

ChatGPT-3.5: This column represents the overall average CodeBlue score obtained from ChatGPT-3.5 across all evaluation instances.

Google Bard: This column displays the overall average CodeBlue score obtained from Google Bard across all evaluation instances.

Android Studio Bot: This column shows the overall average CodeBlue score obtained from Android Studio Bot across all evaluation instances.

The table provides a comparative analysis of the effectiveness of different LLMs in repairing Android Java code vulnerabilities based on their overall CodeBlue scores.

Additionally, the "Overall Average of CodeBlue Score" row presents the average CodeBlue score calculated across all LLMs, offering a summary of the combined performance of these tools in vulnerability repair.

This table facilitates easy comparison and interpretation of the overall effectiveness of the LLMs in addressing Android Java code vulnerabilities.

The visual representation in Figure 5.4 illustrates the overall BleuScore of three different Large Language Models (LLM Models). This graphical representation provides a clear comparison of the BleuScore performance across the ChatGPT-3.5, Google Bard, and Android Studio Bot models. By combining both the graphical and tabular representations, we gain a comprehensive understanding of how these LLM Models compare in terms of their overall BlueScore performance.

5.2.3 HUMAN EVALUATION MATRIX

In the Human Evaluation Matrix presented in Table 5.5, we delve into a crucial aspect of assessing the quality of predicted code by comparing it against the correct code. Each prediction is evaluated and categorized as either 'Correct' or 'Incorrect,' denoted by binary values of 0 and 1, respectively.

For each of the evaluated Large Language Models (LLM Models), namely ChatGPT-3.5, Google Bard, and Android Studio Bot, we tabulate the number of correct predictions ('Correct'), the number of incorrect predictions ('Incorrect'), and the total number of predictions made ('Total'). Additionally, we calculate the percentage of correct predictions, the percentage of incorrect predictions, and the overall percentage of predictions for each model.

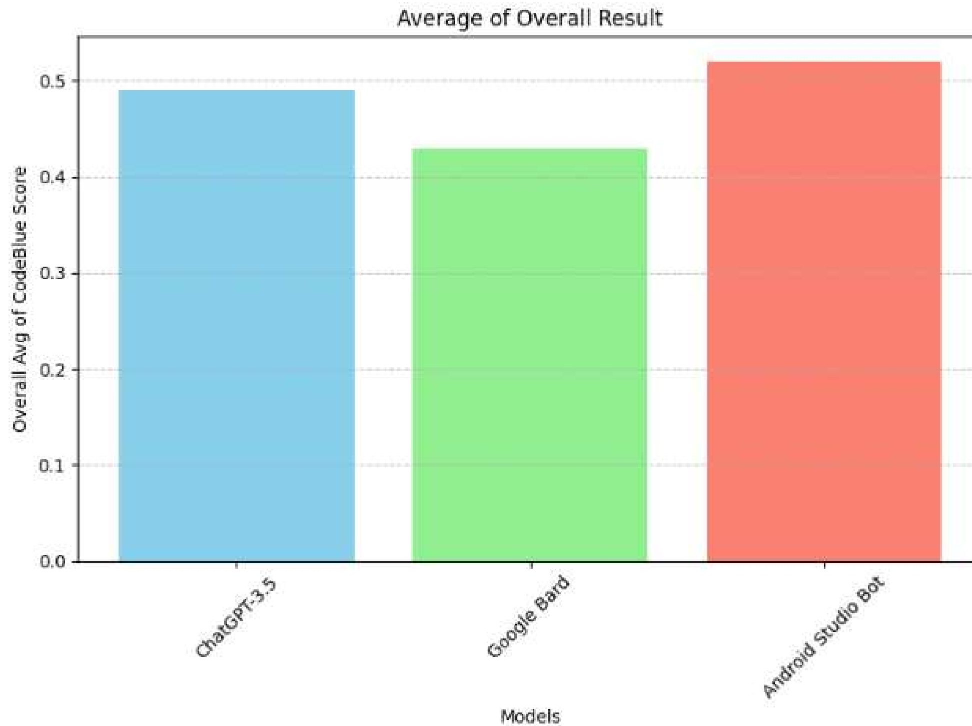

Figure 5.4: Graphic Representation of Overall BlueScore of three LLM Models

	Correct	Incorrect	Total	Correct (%)	Incorrect (%)	Total (%)
ChatGPT-3.5	41	39	80	51%	49%	100%
Google Bard	34	46	80	43%	57%	100%
Android Studio Bot	33	47	80	41%	59%	100%

Table 5.5: Human Evaluation Results

This table 5.5 presents the results of human evaluation conducted to assess the accuracy of predicted code generated by different Language Model-based tools, including ChatGPT-3.5, Google Bard, and Android Studio Bot.

Correct: This column indicates the number of instances where the predicted code matched the correct code as per human evaluation.

Incorrect: This column displays the number of instances where the predicted code did not match the correct code as per human evaluation.

Total: This column represents the total number of evaluation instances considered for each Language Model-based tool.

Correct (%): This column calculates the percentage of instances where the predicted code was deemed correct by human evaluators out of the total evaluation instances.

Incorrect (%): This column calculates the percentage of instances where the predicted code was deemed incorrect by human evaluators out of the total

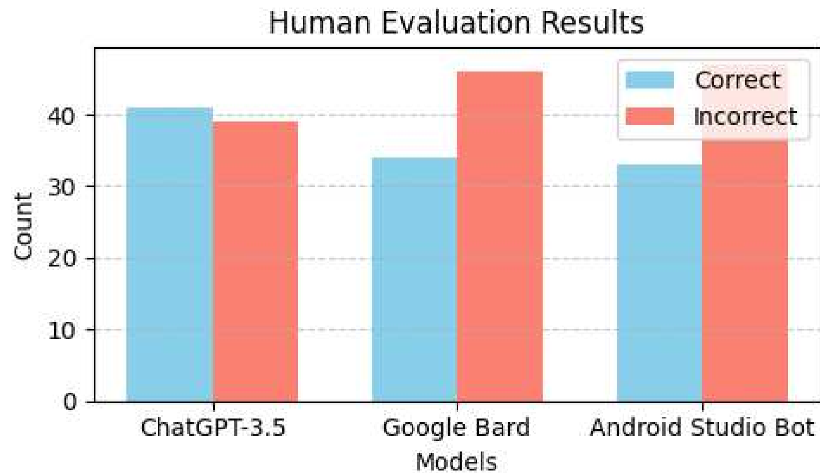

Figure 5.5: Graphic Representation of Overall Human Evaluation of three LLM Models

evaluation instances.

Total (%): This column represents the overall percentage of evaluation instances considered out of the total number of instances.

The table provides insights into the accuracy of predicted code generated by each Language Model-based tool based on human evaluation. It showcases the proportion of correct and incorrect predictions and allows for a comparative analysis of their performance in accurately repairing Android Java code vulnerabilities. The graphic representation of overall human evaluation of the three LLM Models is visually depicted in Figure 5.5. These human evaluation results complement the automated evaluation metrics presented earlier, offering a comprehensive assessment of the effectiveness of Language Model-based tools in vulnerability repair.

5.3 Overview of Results

The evaluation process aimed to assess the effectiveness of three LLM-based toolsChatGPT, Google Bard, and Android Studio Botfor automated Android security vulnerability repair.

5.3.1 Quantitative Metrics

In this section, we present quantitative metrics comparing different language model (LLM) systems. The table (5.6) displays the Blue Score Evaluation Matrix and Human Evaluation results for three LLM models: ChatGPT, Google Bard, and Android Studio Bot. These metrics provide insights into the performance of each model in generating human-like text responses.

LLM Models	Blue Score Evaluation Matrix	Human Evaluation
Chatgpt	49%	41
Google Bard	43%	34
Android Studio Bot	53%	33

Table 5.6: Overall Evaluation Results

Additionally, a visual representation of the overall evaluation results for correcting vulnerable code by the LLM systems is provided in Figure 5.6. This graphic complements the numerical data presented in the table, offering a comprehensive view of the models' performance in fixing vulnerable code.

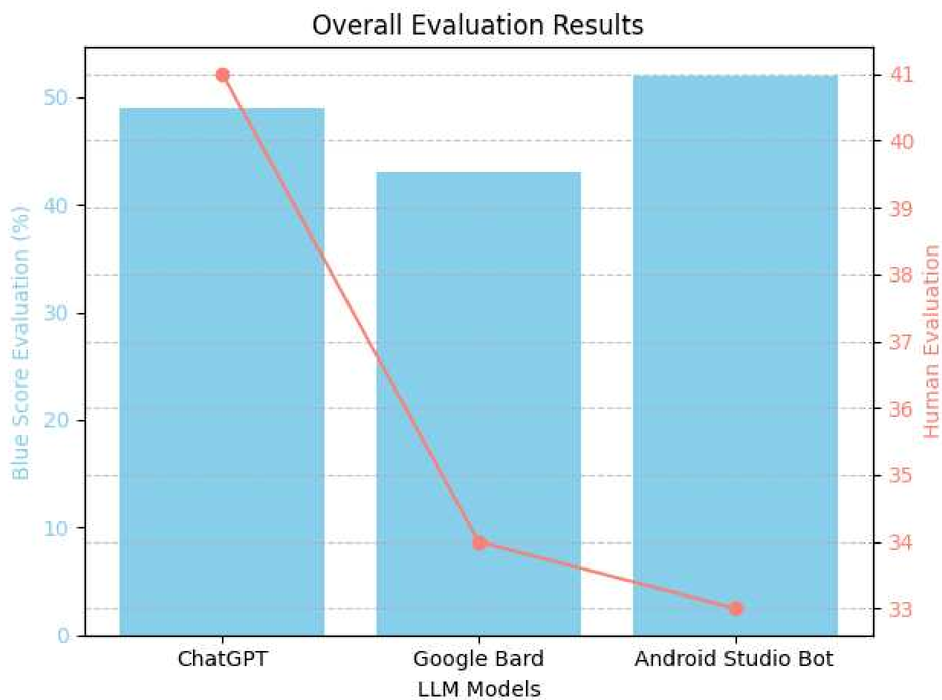

Figure 5.6: Overall Evaluation Result Correct fix vulnerable code

Conclusions and Future Works

6.1 Conclusion

In this study, we conducted a comprehensive evaluation of Language Model (LLM)-based tools for automated Android security vulnerability repair. The evaluation encompassed the analysis of existing tools, empirical assessments of their performance, and comparative analysis of their strengths and weaknesses. Based on our findings, we draw the following conclusions:

The absence of an authentic benchmark or dataset dedicated to Android security bulletins poses a significant challenge in the realm of Android development, particularly for Language Model (LLM) applications. Addressing this deficiency head-on, our thesis leverages Google Android Security Bulletins as the sole credible source of vulnerabilities accompanied by their respective solutions. By meticulously collecting data from this repository, we not only bridge a critical gap in Android vulnerability research but also establish a robust foundation for evaluating LLM-based tools.

However, the absence of standardized evaluation metrics compounds the challenge. In the absence of a systematic and syntactically accurate evaluation matrix tailored for assessing code repairs, we turn to BLEU score as a pragmatic albeit imperfect solution. While BLEU score may not comprehensively capture semantic nuances, its widespread adoption within the field underscores its utility as a benchmarking tool. Furthermore, recognizing the limitations of automated metrics, we complement our evaluation with manual assessments to ensure the integrity and validity of our findings.

In our evaluation, we observed varying degrees of performance among the LLMs under scrutiny. Specifically, the BLEU scores for ChatGPT, Google Bard, and Android Studio Bot were found to be 49%, 43%, and 52%, respectively. These scores, while indicative, do not paint a complete picture of the models' capabilities. Thus, to augment our assessment, we conducted manual evalua-

tions wherein human experts meticulously scrutinized the output of each LLM against manually verified fixes. Notably, for ChatGPT, the human evaluation yielded a score of 41, indicating areas for improvement, while Google Bard received a score of 34, showcasing its proficiency in certain contexts. Conversely, Android Studio Bot garnered a score of 33, highlighting its efficacy in some scenarios.

In essence, our study serves as a pioneering endeavor in the domain of LLM-based automated security measures for Android applications. By addressing critical lacunae in benchmarking and evaluation methodologies, we not only advance the state-of-the-art but also lay the groundwork for future research endeavors. Our reliance on Google Android Security Bulletins for data and the judicious combination of quantitative and qualitative evaluation methodologies underscore the rigor and authenticity of our approach. Moving forward, our findings not only inform the development of more effective LLM-based tools but also pave the way for standardized evaluation frameworks tailored to the unique challenges posed by Android security vulnerabilities.

6.2 Future Direction

To propel the field of automated Android security vulnerability repair forward, future research should prioritize several key directions. Firstly, there is a need to explore hybrid repair approaches that integrate both static and dynamic analysis techniques. By combining the strengths of these methods, researchers can develop more comprehensive and effective solutions for identifying and remedying vulnerabilities in Android applications. Secondly, the development of domain-specific repair models tailored to the unique characteristics of Android application development is essential. These models should be equipped to handle the intricacies of the Android ecosystem, including its diverse architecture and security challenges. Additionally, investigating adversarial robustness and model fairness in Language Model (LLM)-based tools is crucial to ensure their reliability and integrity in real-world scenarios. Lastly, there is a pressing need to evaluate repair techniques in actual development environments to gauge their practical effectiveness and usability. By addressing these research directions, the field can advance the state-of-the-art in Android security and fortify mobile applications against emerging cyber threats. Moreover, to facilitate the work of researchers and developers, establishing a comprehensive benchmark is imperative. This benchmark should systematically compile data related to Common Vulnerabilities and Exposures (CVEs), including CVE codes, associated data stores, complete code samples, and ground truth results. Centralizing this data into a user-friendly repository will streamline research and development efforts in cybersecurity, empowering users to effectively analyze and utilize CVE-related information. Additionally, to enhance the performance of Language Models (LLMs) for addressing Android security vulnerabilities, a sys-

tematic approach involves updating the model with a large dataset specifically focusing on Android vulnerabilities. This dataset should encompass diverse CVEs associated with Android, detailed contextual information, and relevant code samples. Thorough training, testing, and documentation of the model's development processes are essential to ensure its effectiveness and reliability. Furthermore, refining the model's architecture and training methodologies and conducting rigorous testing against ground truth results are imperative to validate its capabilities. By fostering collaboration and transparency between researchers and developers, these initiatives aim to cultivate a robust ecosystem for addressing Android security vulnerabilities, ultimately enhancing the security and resilience of Android applications and systems.

Appendix

In this Section wants to illustrate examples of vulnerable code and respective ground truth for and also the fix generated by the LLM Models with that respective cve and information.

7.1 Example 1

CVE: A-242703780
Description: In the AutomaticZenRule class of AutomaticZenRule.java, there exists a potential failure to persist permissions settings due to resource exhaustion. This vulnerability could lead to local escalation of privilege without requiring additional execution privileges. Notably, user interaction is not needed for exploitation.

The vulnerable code snippet (see Code 7.1) is corrected code is presented in 7.2. Fixes generated by ChatGPT are provided in 7.3, while fixes generated by Google Bard are presented in 7.5. Additionally, fixes generated by Android Studio Bot are available in 7.4. These implementations address the security flaw identified by CVE A-242703780.

```
1 public AutomaticZenRule(@NonNull String name,
  @Nullable ComponentName owner,
2             @Nullable ComponentName
  configurationActivity, @NonNull Uri conditionId,
3             @Nullable ZenPolicy policy, int
  interruptionFilter, boolean enabled) {
4         this.name = name;
5         this.owner = owner;
6         this.configurationActivity =
  configurationActivity;
7         this.conditionId = conditionId;
8         this.interruptionFilter = interruptionFilter;
9         this.enabled = enabled;
10        this.mZenPolicy = policy;
11    }
```

Code 7.1: Vulnerable code of CV A-242703780

```
1 public AutomaticZenRule(@NonNull String name, @Nullable
  ComponentName owner,
2             @Nullable ComponentName configurationActivity
  , @NonNull Uri conditionId,
3             @Nullable ZenPolicy policy, int
  interruptionFilter, boolean enabled)
4 {
5         this.name = getTrimmedString(name);
6         this.owner = getTrimmedComponentName(owner);
7         this.configurationActivity =
  getTrimmedComponentName(configurationActivity);
8         this.conditionId = getTrimmedUri(conditionId);
9          this.interruptionFilter = interruptionFilter;
10         this.enabled = enabled;
11         this.mZenPolicy = policy;
12 }
```

Code 7.2: Correct code of CV A-242703780

```
1 public AutomaticZenRule(@NonNull String name, @Nullable
  ComponentName owner,
2         @Nullable ComponentName configurationActivity,
  @NonNull Uri conditionId,
3         @Nullable ZenPolicy policy, int interruptionFilter
  , boolean enabled) {
4     this.name = name;
5     this.owner = owner;
6     this.configurationActivity = configurationActivity;
```

```
7       this.conditionId = conditionId;
8       this.mZenPolicy = policy; // Corrected order
9       this.interruptionFilter = interruptionFilter;
10      this.enabled = enabled;
11  }
```

Code 7.3: Fix Generated by ChatGPT of CV A-242703780

```
public AutomaticZenRule(@NonNull String name, @Nullable
    ComponentName owner,
            @Nullable ComponentName configurationActivity,
     @NonNull Uri conditionId,
            @Nullable ZenPolicy policy, int
    interruptionFilter, boolean enabled) {
        this.name = name;
        this.owner = owner;
        this.configurationActivity = configurationActivity
    ;
        this.conditionId = conditionId;
        this.interruptionFilter = interruptionFilter;
        this.enabled = enabled;
        this.mZenPolicy = policy;
    }
```

Code 7.4: Fix Generated by Android Studio Bot of CV A-242703780

```
public AutomaticZenRule(@NonNull String name, @Nullable
    ComponentName owner,
            @NonNull ComponentName configurationActivity,
    @NonNull Uri conditionId,
            @Nullable ZenPolicy policy, int
    interruptionFilter, boolean enabled) {
        this.name = name;
        this.owner = owner;
        this.configurationActivity = configurationActivity
    ;
        this.conditionId = conditionId;
        this.interruptionFilter = interruptionFilter;
        this.enabled = enabled;
        this.mZenPolicy = policy;
    }
```

Code 7.5: Fix Generated by Google bard of CV A-242703780

7.2 Example 2

CVE: 20222489
Description: HDCP: buffer over flow check

The vulnerable code snippet (see Code 7.6) is corrected code is presented in 7.7. Fixes generated by ChatGPT are provided in 7.8, while fixes generated by Google Bard are presented in 7.10. Additionally, fixes generated by Android Studio Bot are available in 7.9. These implementations address the security flaw identified by CVE-20222489.

```
case HDCP_ENCRYPT:
        {
            size_t size = data.readInt32();
            void *inData = malloc(2 * size);
            void *outData = (uint8_t *)inData + size;
}
```

Code 7.6: Vulnerable code of CVE-20222489

```
case HDCP_ENCRYPT:
{
    size_t size = data.readInt32();
size_t bufSize = 2 * size;
     void *inData = NULL;
            if (bufSize > size) {
                inData = malloc(bufSize);
            }

            if (inData == NULL) {
                reply->writeInt32(ERROR_OUT_OF_RANGE)
   ;
                return OK;
            }

    void* outData = (uint8_t*)inData + size;
```

Code 7.7: Correct code of CVE-20222489

```
case HDCP_ENCRYPT:
{
    size_t size = data.readInt32();
    if (size > (SIZE_MAX / 2)) {
        return ERROR_BUFFER_OVERFLOW;
    }
    void* inData = malloc(2 * size);
    if (inData == nullptr) {
        return ERROR_MEMORY_ALLOCATION_FAILED;
    }
    void* outData = (uint8_t*)inData + size;
    free(inData);
}
```

Code 7.8: Fix Generated by ChatGPT of CVE-20222489

```
case HDCP_ENCRYPT:
        {
            size_t size = data.readInt32();
            if (size > MAX_HDCP_BUFFER_SIZE) {
                return ERROR_OUT_OF_RANGE;
            }
            void *inData = malloc(2 * size);
            void *outData = (uint8_t *)inData + size;
```

Code 7.9: Fix Generated by Android Studio Bot of CVE-20222489

```
case HDCP_ENCRYPT:
{
  size_t size = data.readInt32();
  if (size > SIZE_MAX) {
    return ERROR_INVALID_SIZE;
  }
  void* inData = malloc(2 * size);
  if (inData == nullptr) {
    return ERROR_OUT_OF_MEMORY;
  }
  void* outData = static_cast<uint8_t*>(inData) +
   size;
}
```

Code 7.10: Fix Generated by Google Bard of CVE-20222489

7.3 Example 3

CVE: 202144228
Description: Apache Log4j2 versions 2.0-beta9 through 2.15.0 (excluding security releases 2.12.2, 2.12.3, and 2.3.1) have a vulnerability related to JNDI features used in configuration, log messages, and parameters. These features do not protect against attacker-controlled LDAP and other JNDI-related endpoints. An attacker who can control log messages or log message parameters can execute arbitrary code loaded from LDAP servers when message lookup substitution is enabled. Starting from log4j version 2.15.0, this behavior has been disabled by default, and from version 2.16.0 (along with 2.12.2, 2.12.3, and 2.3.1), this functionality has been completely removed. Note that this vulnerability is specific to log4j-core and does not affect log4net, log4cxx, or other Apache Logging Services projects [16]

The vulnerable code snippet (see Code 7.11) is corrected code is presented in 7.12. Fixes generated by ChatGPT are provided in 7.13, while fixes generated by Google Bard are presented in 7.15. Additionally, fixes generated by Android Studio Bot are available in 7.14. These implementations address the security flaw identified by CVE-202144228.

```
1 public List<ActivityManager.RunningAppProcessInfo>
  getRunningAppProcesses() {
2         enforceNotIsolatedCaller("
  getRunningAppProcesses");
3         List<ActivityManager.RunningAppProcessInfo>
  runList = null;
4         final boolean allUsers = ActivityManager.
  checkUidPermission(INTERACT_ACROSS_USERS_FULL,
5                 Binder.getCallingUid()) ==
  PackageManager.PERMISSION_GRANTED;
6         int userId = UserHandle.getUserId(Binder.
  getCallingUid());
7         synchronized (this) {
8             for (int i=mLruProcesses.size()-1; i>=0;
  i--) {
9                 ProcessRecord app = mLruProcesses.get
  (i);
10                if (!allUsers && app.userId != userId
  ) {
11                    continue;
12                }
13                }
14                }
```

```
15                  }
```

Code 7.11: Vulnerable code of CVE-202144228

```
1 public List<ActivityManager.RunningAppProcessInfo>
   getRunningAppProcesses() {
2     enforcePermission(permission.REAL_GET_TASKS,
   Binder.getCallingPid(), Binder.getCallingUid());
3 final int callingUid = Binder.getCallingUid();
4     List<ActivityManager.RunningAppProcessInfo>
   runList = null;
5     final boolean allUsers = ActivityManager.
   checkUidPermission(INTERACT_ACROSS_USERS_FULL,
6                callingUid) == PackageManager.
   PERMISSION_GRANTED;
7         final int userId = UserHandle.getUserId(
   callingUid);
8         final boolean allUids = isGetTasksAllowed(
9                 "getRunningAppProcesses", Binder.
   getCallingPid(), callingUid);
10 synchronized (this) {
11         for (int i = mLruProcesses.size() - 1; i >=
   0; i--) {
12             ProcessRecord app = mLruProcesses.get(i);
13             if ((!allUsers && app.userId != userId)
14                         || (!allUids && app.uid !=
   callingUid)) {
15                 continue;
16             }
17         }
18     }
19 }
```

Code 7.12: Correct code of CVE-202144228

```
1 public List<ActivityManager.RunningAppProcessInfo>
   getRunningAppProcesses() {
2     enforcePermission(permission.REAL_GET_TASKS,
   Binder.getCallingPid(), Binder.getCallingUid());
3
4     List<ActivityManager.RunningAppProcessInfo>
   runList = null;
5     final boolean allUsers = ActivityManager.
   checkUidPermission(INTERACT_ACROSS_USERS_FULL,
```

```
6               Binder.getCallingUid()) == PackageManager
    .PERMISSION_GRANTED;
7     int userId = UserHandle.getUserId(Binder.
    getCallingUid());
8     synchronized (this) {
9         // Iterate across all processes
10        for (int i = mLruProcesses.size() - 1; i >=
    0; i--) {
11            ProcessRecord app = mLruProcesses.get(i);
12            if (!allUsers && app.userId != userId) {
13                continue;
14            }
15        }
16    }
17 }
```

Code 7.13: Fix Generated by ChatGPT of CVE-202144228

```
1 public List<ActivityManager.RunningAppProcessInfo>
    getRunningAppProcesses() {
2         enforceNotIsolatedCaller("
    getRunningAppProcesses");
3         enforceCallingPermission(REAL_GET_TASKS, "
    getRunningAppProcesses");
4         // Lazy instantiation of list
5         List<ActivityManager.RunningAppProcessInfo>
    runList = null;
6         final boolean allUsers = ActivityManager.
    checkUidPermission(INTERACT_ACROSS_USERS_FULL,
7                         Binder.getCallingUid()) ==
    PackageManager.PERMISSION_GRANTED;
8         int userId = UserHandle.getUserId(Binder.
    getCallingUid());
9         synchronized (this) {
10             // Iterate across all processes
11             for (int i=mLruProcesses.size()-1; i>=0;
    i--) {
12                 ProcessRecord app = mLruProcesses.get
    (i);
13                 if (!allUsers && app.userId != userId
    ) {
14                     continue;
15                 }
16 }
17 }
18 }
```

Code 7.14: Fix Generated by Android Studio Bot of CVE-202144228

```
1 public List<ActivityManager.RunningAppProcessInfo>
    getRunningAppProcesses() {
2   enforceNotIsolatedCaller("getRunningAppProcesses");
3
4   if (ActivityManager.checkUidPermission(permission.
    REAL_GET_TASKS, Binder.getCallingUid()) !=
    PackageManager.PERMISSION_GRANTED) {
5     throw new SecurityException("Insufficient
    permission: Requires permission.REAL_GET_TASKS");
6   }
7
8   List<ActivityManager.RunningAppProcessInfo> runList
     = null;
```

```
9    final boolean allUsers = ActivityManager.
     checkUidPermission(INTERACT_ACROSS_USERS_FULL,
10       Binder.getCallingUid()) == PackageManager.
     PERMISSION_GRANTED;
11   int userId = UserHandle.getUserId(Binder.
     getCallingUid());
12
13   synchronized (this) {
14     for (int i = mLruProcesses.size() - 1; i >= 0; i
     --) {
15       ProcessRecord app = mLruProcesses.get(i);
16       if (!allUsers && app
```

Code 7.15: Fix Generated by Google Bard of CVE-202144228

References

[1] Larissa Azevedo, Altino Dantas, and Celso G. Camilo-Junior. "Droid-Bugs: An Android Benchmark for Automated Program Repair". In: *arXiv* abs/1809.07353 (Oct. 2018).

[2] Bhargav Nagaraja Bhatt and Carlo A. Furia. "PlumbDroid: Automated Detection and Fixing of Resource Leaks in Android Applications". In: *Journal of Systems and Software* 192.111417 (2024), pp. 111417–111428.

[3] Islem Bouzenia et al. *TraceFixer: Execution Trace-Guided Program Repair*. Accessed: 2024-03-27. 2024. URL: https://arxiv.org/abs/2304.12743.

[4] Creative Commons. *Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)*. Year. URL: https://creativecommons.org/licenses/by-sa/3.0/.

[5] D. "Understanding Transformer Model Architectures". In: *Practical Artificial Intelligence* (Feb. 2023). [Online; accessed 15-March-2024]. URL: https://www.practicalai.io/understanding-transformer-model-architectures/.

[6] Deepset. *Beginner's Guide to Leveraging Large Language Models (LLMs) for Prompting*. URL: https://haystack.deepset.ai/blog/beginners-guide-to-llm-prompting.

[7] Hugging Face. *Hugging Face Transformers Documentation: Prompting*. URL: https://huggingface.co/docs/transformers/tasks/prompting.

[8] Ajay Kumar Jha, Mohayeminul Islam, and Sarah Nadi. "JTESTMIGBENCH and JTESTMIGTAX: A benchmark and taxonomy for unit test migration". In: *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Institute of Electrical and Electronics Engineers (IEEE). 2023, pp. 1–8. DOI: https://doi.org/10.1109/saner56733.2023.00077.

[9] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. "Characterizing Android-specific Crash Bugs". In: *Proceedings of the 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)* (2019), pp. 1–10. DOI: https://doi.org/10.1109/mobilesoft.2019.00024.

[10] Nan Jiang et al. *Impact of Code Language Models on Automated Program Repair*. Accessed: 2024-03-27. 2024. URL: https://arxiv.org/abs/2302.05020.

[11] Nan Jiang et al. *KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair*. Accessed: 2024-03-27. 2024. URL: https://arxiv.org/abs/2302.01857.

[12] Sungmin Kang et al. *Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction*. Accessed: 2024-03-27. 2024. URL: https://arxiv.org/abs/2209.11515.

[13] Maria Kechagia et al. “Evaluating Automatic Program Repair Capabilities to Repair API Misuses”. In: *IEEE Transactions on Software Engineering* 48.7 (2022), pp. 2658–2679. DOI: 10.1109/tse.2021.3067156.

[14] Mario Linares-Vásquez et al. “Enabling Mutation Testing for Android Apps”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017. DOI: 10.1145/3106237.3106275.

[15] Muness. *Leveraging Large Language Models (LLMs) for Prompt Types*. URL: https://muness.com/posts/llm-prompt-types/.

[16] *NVD - CVE-2021-44228*. https://nvd.nist.gov/vuln/detail/CVE-2021-44228. Accessed: 2024-03-16.

[17] Shuo Ren et al. “CodeBLEU: a Method for Automatic Evaluation of Code Synthesis”. In: *arXiv preprint arXiv:2009.10297* (2020). Available at https://doi.org/10.48550/arXiv.2009.10297.

[18] Check Point Research. “Security Analysis of Android Applications in 2023”. In: *Check Point Research Report* (2023).

[19] Oliviero Riganelli et al. “A Benchmark of Data Loss Bugs for Android Apps”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (2019), p. 5.

[20] Ahmed Sherif. *Global market share held by mobile operating systems since 2009*. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/. [Accessed: March 26, 2024]. 2022.

[21] Analytics Vidhya. *Inner Workings of LLMs*. July 2023. URL: https://www.analyticsvidhya.com/blog/2023/07/inner-workings-of-llms/.

[22] Wikipedia. *Transformer (deep learning architecture) — Wikipedia, The Free Encyclopedia*. [Online; accessed 15-March-2024]. 2022. URL: https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture).

[23] B. Young. *26 Tips for Prompt Engineering at Every Model Size*. Year. URL: https://wandb.ai/byyoung3/ml-news/reports/26-Tips-for-Prompt-Engineering-at-Every-Model-Size---Vmlldzo2NDA5NDgx.