UNIVERSITÀ DEGLI STUDI DI PADOVA

DEPARTMENT OF MATHEMATICS "TULLIO LEVI-CIVITA"

MASTER THESIS IN CYBERSECURITY

LLMS EVALUATION FOR ANDROID VULNERABILITY REPAIR

SUPERVISOR
PROF. ELEONORA LOSIOUK
UNIVERSITÀ DEGLI STUDI DI PADOVA

MASTER CANDIDATE
ELISA BRACONARO

STUDENT ID
2057063

ACADEMIC YEAR
2022-2023

“Stay Grateful.”

Acknowledgments

I would like to thank my supervisor, Prof. Eleonora Losiouk, for guiding me and trusting me from the beginning of this journey. For addressing all my questions with seriousness and kindness, and giving me the opportunity to extend my knowledge and experiences.

Abstract

Automated Program Repair (APR) aims to automate the process of fixing software defects, including syntactic errors, semantic bugs, and vulnerabilities. Emergent studies have explored the use of Large Language Models (LLMs) for APR related tasks, such as code refinement and fault localisation, but they are mainly involved in the repair of syntax and semantic bugs. The specific focus on vulnerability repair, especially in the context of Android applications, remains limited. This project wants to take a step further in this direction, by contributing a new dataset of real-world Android vulnerabilities and corresponding fixes and by evaluating three prominent LLMs (ChatGPT-3.5, Google Bard, and Android Studio Bot) on such a dataset in a zero-shot scenario, to understand their actual abilities in repair Android vulnerabilities. Notably, although Android Studio Bot is strictly related to the Android Studio platform and should facilitate Android code generation, its performance for the specific vulnerability repair task is to be considered unsatisfactory, letting ChatGPT-3.5 and Google Bard stand out.

Contents

Listing of figures

Listing of tables

1

Introduction

Automated Program Repair (APR) is a widely recognised research field, whose goal is to automate the process of fixing software defects, commonly known as *bugs*. Their complexity may vary, starting from syntax errors which can be easy and quick to fix, to semantic errors that may provoke unexpected behavior or logical inconsistencies in the faulty program. More importantly, another type of software defect may be introduced by developers: undesired vulnerabilities, which most of the time are not suddenly identifiable.

The reason why APR was born is tightly related to the idea - *and the need* - of having automated tools coming to help developers in identify defects and fix them. The process of localising the defect and then know exactly how to remove it, is in most cases expensive (both at time spent and resources needed level). APR techniques evolved during time, initially based on heuristics, they have been improved as soon as new technologies were proposed. This is the case of Large Language Models (LLMs), a completely new way to interact with machines which is considered revolutionary. From the moment they emerged, many different research fields started to explore their abilities and limits, and the same applies for APR. Thanks to the very large amount of data on which LLMs are trained, their knowledge covers many and different knowledge domains.

Considering the world of automated program repair, plenty of researches are carried out to answer one common question: ”*How good are LLMs in repairing software defects?*”. Several studies have been and are being published, trying to give an answer from various point of views, such as syntax and semantic errors repairs, multi-language repair and so on, but one particular aspect remains almost uncovered: vulnerability repair, with or without the employment of large language models.

Many of APR related studies began to emerge from the early 2000s and started from the basis, trying to develop tools which were able to repair standard syntactic errors, moving then towards the fixing of semantic bugs, but, although much has been achieved, there are still challenges to overcome and vulnerability repair is one of them. Only in recent years, the problem of fixing vulnerabilities has been separated from the one related to repair logic errors, and new studies have been proposed [1], [2], but still not so many with respect to syntactic and semantic problems.

Considering that the problem of repairing vulnerability is a crucial aspect to cover and that the investigation about LLMs is a central topic, this project wants to put the two aspects together: "*How good are already existing LLMs in repairing vulnerabilities?*". Specifically speaking, the vulnerabilities taken under analysis come from real-world Android applications. Reasons why this type of vulnerabilities has been chosen as the core of the project are the following: (1) most of traditional APR tools are developed and tested on datasets collecting bugs from Java, C/C++ and Python in the majority of the cases (and only few of them contain vulnerabilities); (2) considering that LLMs started to arise during the very last years and have been trained on most of the available digital knowledge, and considering that the datasets used to test them are not so recent, it is very likely that such datasets belong to their training data and, therefore, are known to the LLMs; (3) at last, there are very few studies in which LLMs are involved to solve Android-related problems, and none of them explore the abilities of LLMs to fix Android vulnerabilities.

For these reasons, the contributions of this project are:

- the realisation of **a new dataset** of Android **vulnerabilities** extracted from real-word applications and corresponding **fixes**; in this way, the dataset can be used for validating both new Android security tools and new APR tools focusing on Android-related defects;

- three existing **LLMs** (ChatGPT-3.5, Google Bard and Android Studio Bot) are **tested** on the new dataset in a *zero-shot* scenario, so that their abilities can be evaluated on examples they never seen (at least, not identical and included in their training set);

- a general understanding of **LLMs performances in repairing Android vulnerabilities**.

In particular, this project evaluate ChatGPT-3.5, Google Bard and Android Studio Bot, three LLMs that are free and easy to use. Moreover, considering the idea of a developer who need to find a suggestion on how repair some vulnerabilities found in his or her code, it is very likely that he or she is going to use an interactive LLM, such as the ones chosen for this analysis. While Android Studio Bot is tightly related to the Android Studio platform and specifically designed with the aim of helping developers to generate Android code, it has poor performances with respect to ChatGPT-3.5 and Google Bard which are, on the contrary, general-purposes LLMs.

The rest of this thesis is divided as follows: Chapter 4 describes all the steps necessary for the creation of the new dataset, including the collection of the applications to be analysed, the detection of related vulnerabilities and the manual generation of the fixes which serve as ground truth. Chapter 5 provides a general understanding of the LLMs chosen for the analysis, how the prompt is structured and how the corresponding outputs are pre-processed before evaluation against the ground truth. Chapter 6 describes the metrics employed to evaluate the models and contain the obtained results. Chapter 7 provides a discussion about the overall evaluation, underlying the limits and the abilities of the analysed models. Chapter 8 includes future directions for this project. Chapter 3 briefly describes related studies with respect to the core topics of this thesis: APR, vulnerability repair, LLMs, Android. Lastly, Chapter 9 concludes the thesis.

2

Background

2.1 AUTOMATED PROGRAM REPAIR (APR)

In recent years, software development went through a series of changes and improvements, which lead not only to a higher quality of software delivered, but also to a consistent increase in complexity of such products. The software development cycle is not free of challenges and software defects are often around the corner. Defects may vary from syntactic errors, due to a bad usage of the programming language, to semantic errors, which result in flaws in the program logic, to the worst scenario which includes critical vulnerabilities. In this regard, Automated Program Repair (APR) identifies a set of techniques with the aim of "*automatically find a solution to software bugs without human intervention*" [3]; new researches in this direction have emerged a lot during the last years. Aligned with the progress of APR techniques and research, there are other related research fields such as fault localization and patch assessment. In particular, fault localization is a necessary first step to perform before conducting APR analysis, since it aims to identify the software defect. Techniques on how to do so fall outside the core scope of APR research field [4], but one strategy is to detect faults in programs is to run the code against a set of test cases (test suites) and if at least one of them is failing, then the program is not behaving as expected and therefore the fault can be identified. In this case, the goal of APR is to suggest a fix which allows the whole test suite to pass –

not introducing new bugs. Hence, taking into consideration the process of patch evaluation, the patch can be classified as: candidate, plausible, correct. The candidate patch is the actual generated patch, as result of the modifications to the buggy code gives an input; a plausible patch is a patch which successfully passes all the test cases. Finally, the correct patch is the plausible patch which is considered correct with respect to a particular metric (syntactically and/or semantically equal to the developer's patch for the same buggy code) [5]. In most cases, human validation is the latest step to guarantee the accuracy of the results.

Latest APR techniques are learning-based, meaning that the process to suggest repairs is automated thanks to the integration of Deep Learning models in the tools. Thanks to DL-models capabilities, the development of new tools which overcome limitations of previous techniques is now possible. In detail, the introduction of neural networks models enabled the automated learning process. Starting from tons of samples like pairs of bugs and related fixes, the accuracy of the models continuously improved allowing better suggested patches thanks to a more accurate prediction. Capabilities of such models go now beyond what expected and this is the case of Large Language Models (LLMs). Leveraging their abilities in generating code and considering their huge knowledge in many different fields, LLMs can be tested and evaluated to perform tasks typical of the APR world, including code refinement, defect detection and so on. For this reason, the focus of this project is to evaluate large language models in performing repair tasks against Android vulnerabilities.

2.2 Large Language Models (LLMs)

Large Language Models (LLMs) are the latest advancement in the field of Language Modeling, which has the objective of developing models able to enhance the capability of machines in understanding, predicting and generating word sequences following the common rules of human language.

Before coming up with LLMs solutions, many learning-based techniques consider the problem of automated program repair as a Neural Machine Translation (NMT) task, going from a code snippet containing the bug to its fixed version in an automated way, therefore treating the repair task as a translation task. Most common NMT models built for repair are based on an encoder-decoder architecture, which permits not only to map the input (buggy code) to the output (patched code), but also to learn the hidden features of the input. This aspect is fundamental in the APR scenario, since learning hidden representations enables the model to also capture the semantic aspects of the code.

A step forward has been taken by researchers when they start to include Language Modeling in the scope of APR research problem. Early stages of studies about Language Modeling include the development of Neural Language Models (NLMs) and Pre-trained Language Models (PLMs). NLMs focus on optimising the prediction of word sequences produced by Neural Networks (NN) and advancement in this direction resulted in new research to try to consider not the single word but its surrounding context at the same time [6]. In this regard, Transformer architecture [7] is now the most used one when it comes to develop new models, thanks to the self-attention mechanism which allows a model to understand the importance of each word in a sequence – so, in a sentence – and how words are related to each-other, making it possible to highlight their relationships and, therefore, the context.

Main differences between traditional DL models and LLMs include the amount of data on which models are pre-trained, having LLMs pre-trained on an enormous amount of unlabelled data and with a huge number of parameters. The most common architecture for LLMs is indeed based on Transformers, due their capabilities in processing languages. The first widely recognised and used LLM was GPT-2 by OpenAI [8], trained with 1.5 billions of parameters. What characterises the widespread usage of LLMs is their ability to deal with complex tasks and being successful most of the time. They can be fine-tuned on smaller amounts of data, to let them improve their solutions on a specific task. In particular, there are two strategies to evaluate them, using zero-shot or few-shot learning. In the first case, the LLM is tested and used as it was pre-trained; in the other case, few pairs of input-output examples are given to the model before being tested.

Among most common architectures and models used to conduct program repair tasks are T5 architecture [9], CodeT5 [10] model based on it for processing code, CodeBERT [11] and GraphCodeBERT [12], which is the first structure-aware pre-trained model, able to learn how to represent code directly by source code and it is also based on a transformer architecture. Built on these models, tools and studies have been published to adapt the models for APR-related tasks, such as syntax, semantic and vulnerability repair. As an example, AlphaRepair [13] uses CodeBERT as an underlying model to predict the correct fix for a given bug and focuses on repairing bugs at semantic level. AlphaRepair is one of the first studies exploring the usage of LLMs for APR and resulted in outperforming all the traditional APR tools [5].

As of the latest updates, GPT-2 was one of the largest LLMs featuring 1.5 billion of parameters, impressively surpassed by GPT-3 which counts 175 billion of parameters. Updated version GPT-3.5 is fine-tuned using reinforcement learning from human feedback and counts fewer parameters; for GPT-4 the number of used parameters is not yet published. Such a huge number of parameters allow LLMs to understand very complicated patterns, relationships between data and reach results never seen before. In this regard, the major LLMs chosen for evaluating their performance in the field of Android vulnerability repair are ChatGPT-3.5, Google Bard [14] (trained on 137 billion parameters) and Android Studio Bot [15].

2.3 Vulnerability Repair

Vulnerabilities are the most critical type of software defects, since they may lead to possible security breaches and dangerous consequences for both users and companies. However, they are often unconsciously introduced by developers and the repair process can result expensive in terms of time spent to identify them and then apply the patch. Software defects considered as a target for APR techniques are commonly categorised into: syntax errors, programming (semantic) errors and security vulnerability errors. Syntax errors are mainly due to lack of experience and knowledge in programming from developers, but they can be easily fixed by themselves or with the help of new tools, such as DeepFix [16], which can automate the process and repairs multiple bugs, iteratively checking each one against an oracle to assert whether the produced patch can be accepted or not. Programming or semantic errors are raised from a bad implementation of the logic behind the program, but they are not necessarily vulnerabilities. After identifying the bug through the usage of fault localization techniques, and once the patch is generated by an APR tool, if it passes all the test cases in the appropriate test suite, then it can be considered correct and the logic of the program fixed.

Only in recent years, the automation of vulnerability repair has become a separate research problem with respect to normal program repair (including syntactic and semantic errors), due to the realisation that vulnerabilities are more urgent, critical and dangerous to fix than a “normal” bug.

2.4 ANDROID OPERATING SYSTEM

The Android Operating System owned by Google is the most used operating system on smartphones [17]. Its architecture is designed to enhance security and prevent the direct interaction between the user and the kernel space. It is based on the Linux kernel, which has been adapted for mobile devices and enforces mandatory access control (MAC) through the use of SELinux. There are two fundamental principles that each component follow:

1. separation: each application has its own sandbox, and cannot directly communicate with other applications installed on the mobile device. It is defined at process and file-system level, which means that each application runs its own process and cannot access resources from others;

2. least privilege: each application has to declare the minimum number of permissions.

If an application declares permissions for resources it does not need, a malicious application can exploit it to perform harmful actions. Google suggests a series of best practices [18] that should help developers in designing safe and secure applications and that should reduce to the minimum the chances of introducing flaws in their code. However, developers may introduce vulnerabilities in an unconscious way through a poor implementation of important Android components. Providing a short description of them allows one to better understand how an attacker can exploit vulnerabilities targeting these components.

- **Activity**: used to manage graphical user interface; there is a "Main Activity" which runs as soon as the user clicks on the app's icon, but an application can have several activities allowing the user to interact with it through the graphical interface;

- **Service**: do not provide a user interface; a service performs an action in background for a certain amount of time, without interrupting the user interaction with the application;

- **Broadcast receiver**: it is called to respond to system-wide events and perform a consequent action when they are received; a broadcast receiver register to an event and waits for it to happen;

- **Content provider**: it allows applications to share data indirectly through the usage of APIs. When an application needs to expose data, using a content provider ensures that other applications do not have direct access to the data source. They can access this data, if they have the necessary permissions, through the interaction with the corresponding content provider.

If one of these components is badly implemented, vulnerabilities can arise and be exploited. For instance, if an application is defining a new way to interact with its data source creating a new class extending the content provider class, when overriding sensitive methods such as "query()", then depending on how queries are defined, SQL injection may be a risk. Another vulnerability might be caused by exposing "to the world" components managing sensitive data, making them public but not protected. In the Android manifest file, which represents the most important file for an Android application, all the components are defined and permissions declared. If a service has the exported attribute set to "true", then it is visible not only to other components of the same application, but also to every other application. In order to protect it, it may be necessary to add a permission, so that only applications declaring such a permission can interact with the component.

Recalling that each application has its own sandbox and it is completely isolated from all the others, the Android operating system introduced different solutions to make them communicate with each other: inter procedure communication mechanisms (IPC). Among them, there are Intents, messaging objects which simplify the communication between components of the same application and toward another one; they can be declared as implicit or explicit. If explicit, the intent shares data and / or communicates with a target component; on the contrary, if an intent is declared as implicit, it is just specifying which action should be performed when received. It is launched by a component but it is the responsibility of the Android OS to understand which is the actual destination, if more than one component is able to manage such an intent. Intent hijacking may be a consequence of implicit intent: since they are not referring to a specific component, a malicious application may declare a component to be able to manage an intent with a particular action. When the Android OS has to choose which component can handle the intent, it may choose the malicious application, resulting in having an attacker dealing with potentially sensitive data. There are just a few of the many possible vulnerabilities that may arise from a weak implementation. As said in the previous section, vulnerability repair is still a new area to explore and few studies have been conducted on this problem; consequently, a specific sub-field such as repairing Android vulnerability is even more unexplored.

To develop an Android application, it is possible to use both Java and Kotlin languages and additionally, native libraries can be implemented using C or C++. Focusing on this project, the vulnerabilities under analysis are written in the programming language Java. There are some bugs in Android applications which may also fall into the general category of Java vulnerabilities, but along with them, there are others due to specific features of the Android OS.

3

Related Works

3.1 APR TECHNIQUES

The automation of program repair has always been a goal to achieve. Manual fixing of software defects is a common and unavoidable part of the software development process, which takes away a lot of time from developers.

The core objective of program repair is to generate a patch such that the defect is fixed; how the patch is actually generated identifies the main four categories of APR techniques: search- or heuristic-based, constraint-based, template- or pattern-based, learning-based. From the early 2000s several tools started to emerge, which have been then categorised into search-based techniques, since they shared the common strategy in generating patches. The first proposed techniques are referred to search- or heuristic-based, whose underlying idea is to look for the correct patch in a predefined patch domain, using heuristic to generate candidate fixes and using techniques like mutation-selection or test execution to understand which can be a good patch. This type of techniques improved over the years but still present some negative aspects, among which there is an usually easy explosion of the search space [5]. Limitations of these techniques led to the development of constraint-based techniques, and then template-based ones. Consequently, the need to change the way patches are generated derive from such limitations of previous techniques; it was a natural and gradual process during the years, considering the possibility of having new technologies able to take a step over them. For instance, template-based techniques lack continuous learning and have a limited domain of known bug-fix pairs; therefore, new learning-based techniques were born, leveraging the advent of Deep Learning models. Huang, et al. [5] describe the passage from one category to others, referencing the most significant tools for each of them and also addressing their limitations.

Learning-based techniques are the current state of APR development, since they overcome most of the challenges faced by the other techniques, such as a limited knowledge domain, considering both the type of software defects and their programming languages, which are commonly C/C++, Python and Java considering the available benchmarks. Zhang, et al. [19] illustrate common features of learning-based tools for program repair and keep track of most used datasets and metrics for evaluation. The survey also introduces recent advancement in using LLMs for APR, which is the very new direction followed by most studies in APR field at the moment.

A living review about APR techniques is maintained by M. Monperrus [20] and includes all the references to the most relevant studies in the APR research field up to now.

3.2 LLMs for APR

Large language models are a central point in many research fields right now and automated program repair is one of them. Several studies have been recently published and they keep the focus on one main research question, which is to understand how good are LLMs in repairing software defects. Answers to such a question are vary since they depend on the single study and the types of software defects which are under analysis; in most cases, LLMs are fine-tuned to the specific task, either using a small dataset or performing few-shot learning. A general evaluation has been carried out by [21], who tested LLMs performances in three different repair settings: generation of the entire patch, code completion given prefix and suffix and a single line fix. They evaluated several LLMs, including Codex [22], CodeT5, GPT-NEO and others, on common benchmarks used for testing APR tools, which are Defects4J, QuickBugs and ManyBugs. The programming languages covered by these benchmarks are Python, Java and C. One limitation for traditional APR tools is represented by their ability of fixing bugs for only one programming language. Traditional DL models already gave the chance to overcome the limitations of having multi-language tools of which traditional APR tools suffer, but with the employment of LLMs multi-language repair is becoming even easier. The idea of having a single tool which can perform program repair at multi-language level has been explored by another study which proposes RING [23], a new tool based on Codex which outperforms language-specific repair tools for three out of six languages under analysis. Large language models demonstrated to be competitive against traditional APR tools even when no fine-tuning strategy is adopted. The intuition of achieving high performances when using LLMs for APR is confirmed by [24], where Code Language Models (CLMs), like CodeT5 and PLBART, reach a greater number of fixed bugs when fine-tuned; in particular, the best among the one analyses fixes about 70% bugs more than the ones repaired by traditional tools. As in other works, the benchmarks used are always the noted ones (Defects4J, QuickBugs and HumanEval).

Moving even beyond the zero-shot or few-shot scenario, other studies tried another approach which involves a sort of dialogue with the model. To avoid the generation of patches already defined as incorrect, [25] suggest to initiate a ”conversation”, where after prompting the original input, the incorrect generated patch is given as input along with its validation feedback. This solution resulted in being successful, having ChatGPT as the best model in generating patches.

Results coming from these works highlight the potential that LLMs have as repairing tools, outperforming traditional ones most of the time, even more when they are fine-tuned.

3.3 LLMs for Vulnerability Repair

Vulnerability repair is a specific task from the broader scope of automated program repair. Recently, studies are moving towards the implementation of new tools which are precisely targeting the problem of fixing vulnerabilities. VRepair [1] is one of the first approaches in this field; it is pre-trained on a large-scale dataset of bug-fixing samples and then fine-tuned on smaller datasets to gain more knowledge and improve its accuracy on the specific tasks. In particular, LLMs are leveraged also in this scenario. For instance, a new approach, VulRepair [26] which is based on CodeT5, has been evaluated over CWEs against VRepair and CodeBERT and resulted in outperforming its predecessor VRepair and also CodeBERT. Another work [27] focuses on evaluating existent LLMs (Codex, AI21Jurassic [28]) in a zero-shot scenario, to understand how much good are already available LLMs in repairing vulnerabilities coming from programs written in C or Python. LLMs still resulted in being successful.

The specific problem of repairing vulnerabilities is a central topic in new studies, mainly thanks to the capabilities of LLMs, which can facilitate both the vulnerability detection and repair at both levels of zero and few shot learning.

3.4 LLMs for Android

If several studies explored the usage of large language models in many fields, these are not including Android. Feng, et al. [29] propose a lightweight LLM approach, AdbGPT, to perform reproduction of Android bugs using bug reports collected by ReCDroid [30] and ANDROR2+ [31]. Another study [32] leverages GPT models to automate the decision making process for the user interacting with an Android application. However, there are no recent works about Android vulnerability repair through the use of LLMs, which is the scope of this project.

3.5 Android and Vulnerabilities

No recent studies leverage the usage of LLMs for repearing Android vulnerabilities and there are few datasets collecting them. One of this dataset is AndroVul [33], which attempts to put together possible vulnerabilities due to the usage of dangerous permissions, security code smell from the smali code and other possible security related bugs from AndroBugs [34], a testing tool for Android used to parse applications and look for possible security issues. The dataset proposed by AndroVul contains 73 vulnerability metrics extracted from the evaluation of a sample of Android APKs from AndroZoo [35]. In recent years, a tool called Vulvet [36] tried to deal with both detection and repair of Android vulnerabilities but not involving large language models. It focuses on detection of common vulnerabilities and automatically generates patches for some of them, through control-flow instrumentation and reconstruction of methods and parameters. However, patches sometimes require the need of the developer and, therefore they are not completely automated.

4

A new dataset

4.1 MOTIVATIONS

The final goal of this project is to evaluate the performance of Large Language Models (LLMs) in suggesting repairs for vulnerabilities that come from Android applications. To accomplish this, a dataset composed of pairs vulnerability-fix is necessary and there is a lack of datasets covering Android vulnerabilities which also include respective possible fixings. For this reason, one of the main contributions of this project is a new dataset, which collects 284 Android vulnerabilities.

The dataset wants to bring together several vulnerability examples coming from real-world Android applications, covering different categories. To build the dataset, for each vulnerability found, a manually generated fix is proposed and considered as ground truth against which the output of evaluated LLMs will be compared. Reasonings and explanations on how the fixes have been created are described in Section 4.6. Hence, the dataset includes both defects extracted from the manifest file and from actual classes of Android applications, with the corresponding ground truth fix. In Section 4.5.1, all the statistics about data in the new dataset are shown and described.

4.2 REQUIREMENTS AND LIMITATIONS

With the aim of providing a clear explanation of each step made during the realisation of the dataset, it is necessary to start from the initial requirements. Firstly, for each application is necessary to have both the APK and the relative source code. The former is needed for the detection tool, SPECK, described in Section 4.5, while the latter is required for testing the manually generated fix. Since fixes proposed for each vulnerability found are considered as ground truth, it is mandatory to ensure that the application can still be compiled after the integration of the repairs. To respect such a requirement, only applications for which both APK and source code is available have been collected, even if such a choice reflected in having at disposal a significantly reduced number of applications to be analysed.

A consecutive and strictly related challenge is represented by the actual compilation of applications source codes, since most of them, even without the integration of the repair, resulted in having many dependency-related issues when compiling or they do not come with the necessary files for the compilation. However, thanks to the categorization of vulnerabilities which allowed to consequently have a structured format of fixes, such a limitation can be resolved by testing a smaller number of applications, among the ones collected and able to compile, for each type of fix. If a vulnerability comes from an application that cannot be compiled and the same applies for any other application with the same vulnerability, this is implemented in a crafted application, so that the fix proposed as ground truth for that vulnerability type can be tested and assert that no compilation problems due to the integrated repair would arise in the original application.

4.3 DESIGN

The criteria on which the vulnerability detection and extraction is based sees the violation of best practices at its core. Google highlights the guidelines that a developer should implement when building an Android application; the used detection tool is based on them and as soon as an application violates one of this guidelines, the corresponding faulty line is extracted from the decompiled APK. Hence, vulnerabilities collected in the dataset are all violating a defined best practice, identified by a unique number, which is going to be referred to as ”Rule”. All the best practices are described in Appendix B and examples for each of them are included.

The final dataset collects 284 vulnerabilities with the respective ground truth which has been manually generated. It contains three columns:

1. an **id** which is unique and identifies the vulnerability;
2. the **language** of the snippet (Java or XML);
3. the **vulnerability** and its context (e.g. a method);
4. the corresponding **ground truth**.

In addition, to keep track of the application from which the vulnerability is extracted and the specific file (Java class or Android Manifest), another dataset is built and it contains the following information, including the match between the vulnerability extracted from the decompiled APK and its mapping to the source code:

- **id**: unique id which identifies the corresponding vulnerability;
- **apk**: the name of the analysed APK, from which the vulnerable code has been extracted;
- **language**: the programming language of the vulnerability, which may be XML or Java;
- **rule**: a number identifying the violated best practice;
- **severity**: indicates the severity of the vulnerability, which may be ”warning” or ”critical”;
- **dec-path**: the file from the decompiled APK which contains the vulnerability;
- **dec-line-number**: the number of the line introducing the vulnerability in decompiled file;
- **dec-vuln-line**: the vulnerable line in decompiled file;
- **code-path**: the path of the original source code inside the Android project from which the vulnerable snippet has been extracted;
- **line-number**: the number of the line introducing the vulnerability in the source code;
- **vuln-line**: the actual content of the vulnerable line in source code.

Including in the dataset not only pairs composed by vulnerability and fix, but also corresponding information about the application analysed, the specific line number in the source code wants to guarantee that everyone using the dataset can help in reproducing bugs correctly and have a deeper level of information about the specific vulnerability, taking it as a reference.

Sections below will guide through the explanation of each step made to the realisation of the dataset, to understand how it has been constructed, to make clear each decision and illustrate the challenges and limitations faced. To proceed with clarity, the following section describes two remarkable limitations that have led to the adoption of initial necessary assumptions.

4.4 Collection of Android applications

Based on the necessary assumptions explained in Section 4.2, all the applications are downloaded from F-Droid [37], an online repository which is free and open source. It collects applications for different categories, including connectivity, phone & sms, system, security and so on. For every application, both the link to the original source code and the corresponding APK built on it are available. The total number of applications that have been collected and analysed is 138, chosen randomly among the ones available in the repository. Having to satisfy the requirement such that not only it is necessary to get the source code, but it has also to be compiled without any errors, all the applications have been tested for compilation and Table 4.1 shows a summary.

Only the 29% of the applications downloaded, successfully compiled. The other 71% includes applications for which compilation is not possible, due to several problems. This reflects in not having a compilable source code for each vulnerability extracted and for which the corresponding ground truth is not being directly tested into the original application, as described in 4.2.

	failing	no-files	success	total
# apps	40	58	40	138
% apps	29%	42%	29%	100%

Table 4.1: Summary of total number of apps with compilations problems or not.

4.5 VULNERABILITY DETECTION VIA SPECK

The tool used to perform vulnerability detection is SPECK [38]. The core concept which is at the foundation of the tool regards which vulnerabilities are considered and consequently the types of patches proposed. Following Android Security Best Practices defined by Google [18], 32 rules have been defined by the creators of the tool. Each rule represents one best practice that the application should follow and it is implemented in the tool to look for patterns which do not match such a guideline; when a match is found, the line at issue is reported as a vulnerability since it does not implement the best practice. For clarity purposes, an example of best practice definition and related vulnerable code follows:

```java
public void performBackup(ParcelFileDescriptor oldState, BackupDataOutput data,
     ParcelFileDescriptor newState)  {
    Log.d(TAG, "Performing backup");
    ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
    MessageDigest digester = null;
    Writer writer;
    try  {
        digester = MessageDigest.getInstance("MD5");
        writer = new OutputStreamWriter(new DigestOutputStream(byteStream,
    digester),Charset.forName("UTF-8"));
    } catch (NoSuchAlgorithmException e)  {
        writer = new OutputStreamWriter(byteStream, Charset.forName("UTF-8"));
    }
    // rest of the code
}
```

Listing 4.1: Vulnerable Code for Rule 29.

The best practice related to Rule 29 says: ”*Use a recommended cryptographic algorithm if dealing with Cipher, Mac, MessageDigest, Signature objects.*” which is not respected since the algorithm used is *MD5* that cannot be considered as safe as *SHA-256*. (see all best practices with vulnerable code examples in Appendix B).

The workflow of the tool is the following: given an APK as input, it decompiles the APK using JADX [39] and statically analyses the decompiled version, looking for violations. The analysis of the tool is actually a starting point to extract the problematic code, but it is not enough to directly obtain the vulnerabilities to put in the dataset and other steps have been required. Considering that the manually generated fix has to be tested before being declared as ground truth, it is essential to map the vulnerable code found in the decompiled file to the vulnerable line in the project source code. The mapping has been done by using a Python script which finds the match between the two through a similarity score, comparing the extracted vulnerable line to each line of the corresponding file in the project. When performing the mapping, few challenges arised:

- all decompiled classes are written in Java, but this is not always true for the source code, which may be actually written in Kotlin; therefore, a vulnerability found on decompiled file may correspond to a vulnerability in Kotlin which is outside the scope of this project, focused on Android vulnerabilities but only taking into account Java language;
- decompiled code may include obfuscated code, for which is not possible to find the reference code in the source;
- vulnerable code in a decompiled class may be actually included in a class of the source code with a different name;
- since the tool analyse the whole decompiled APK, some violations come by ”external” files such as Java libraries which are reconstructed during the decompilation and naturally, it is not possible to find a match in the source code.

Accordingly, obfuscated code from decompiled classes and vulnerable codes which turned out to be written in Kotlin are excluded from the final dataset. On the other hand, the third point has been bypassed looking for the most similar line in other classes where the decompiled class is not present in the project. Obviously, each match between the vulnerable line in decompiled class to the vulnerable line in source code found by the script has been manually reviewed to be sure of the correctness of the match itself, taking into account the surrounding context.

To produce a correct patch, the output of SPECK is not enough. When an application is analysed and violations found, for each of them the line specifically recalling the violation of the best practice is returned, without any additional context. For this reason, it is necessary to include more context to the line and in most cases, the method in which the faulty line is enough. Thus, the method needed to be extracted from source code for each returned line; such an objective was reached by the development of a script which makes use of libraries defined in the detection tool and adapts them to effectively recover the method starting from a line.

Another step that has been done consisted in filtering all the extracted lines by violation. Not every violations from the ones implemented in the tool lead to vulnerabilities which can be fixed, since:

- some of these best practices can be treated as warnings and as hints for developers; thus, related faulty lines have been excluded from the analysis;
- the implementation of the guidelines needed complex modifications of applications at design-level; related vulnerabilities been excluded, but those will be covered in future.

Moreover, after manually analysing each final vulnerability (from source code and with context) to include in the dataset, some of them resulted to be false positive (i.e. code snippets which do not violate any rule) and others appear more than once, probably due to underlying relationships between collected applications. Hence, since the final goal is to have an original dataset which presents several vulnerabilities but which differs from each other, identical copies of faulty codes have been removed, with the aim of providing then a clean dataset and prompting the LLMs with only prompts different from each other.

4.5.1 SUMMARY OF VULNERABILITIES IN THE DATASET

After having considered the several steps done for detection and extraction of unique vulnerabilities, Table 4.2 keeps a summary of the total number of applications which are included in the final dataset, giving also insights on how many of them presented compilation problems. Taking into account that not every downloaded application presented vulnerabilities which violate the defined best practices, the amount of useful applications is smaller and if also considering that the dataset has been cleaned from identical copies and false positives, only about 50% of the collected applications presented at least one vulnerability which has been included in the dataset.

Almost 75% of the applications for which at least one violation is in the dataset cannot be compiled. This means that for the majority of the vulnerabilities, the proposed ground truth cannot be tested in the original application. However, as described in Section 4.2, this limitation is overcome and a better explanation is provided in Section 4.7.

	failing	no-files	success	total_with_vulns
# apps	28	23	17	68
% apps over total_with_vulns	41.2%	33.8%	25%	100%
% apps over total_collected[1]	20.3%	16.7%	12.3%	49.3%

Table 4.2: Summary of total number of apps with vulnerabilities.

Table 4.3 provides an insights on how many best practices have been violated and a summary of how the vulnerabilities are divided.

rule violated	# vulnerabilities	% vulnerabilities
1	21	7.39%
3	19	6.69%
4	1	0.35%
5	7	2.46%
6	10	3.52%
7	8	2.82%
9	1	0.35%
15	1	0.35%
16	8	2.82%
17	8	2.82%
18	91	32.04%
20	30	10.56%
22	2	0.70%
23	46	16.20%
26	6	2.11%
29	25	8.80%
	284	100 %

Table 4.3: Summary of vulnerability types in the dataset.

---

[1] Total number of applications: 138. See Table 4.1

4.6 Fixes Generation

Once having the complete code snippet incorporating the faulty line, fixes can be developed. The reasoning behind the patch is to correctly implement the related best practice, avoiding the introduction of new vulnerabilities, new bugs or a change in the logic of the application; moreover, the developed patch must let the application compile. As vulnerabilities come in categories, so do the patches. More vulnerable code snippets violating the same best practice are likely to share the same code structure as shown in the two codes below, which are of course extracted from two different applications:

```
/private void previewImageDraft(final @NonNull Slide slide)  {
    if (MediaPreviewActivity.isContentTypeSupported(slide.getContentType()) &&
    slide.getUri() != null)  {
        Intent intent = new Intent(context, MediaPreviewActivity.class);
        intent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
        intent.putExtra(MediaPreviewActivity.SIZE_EXTRA, slide.asAttachment().
    getSize());
        intent.setDataAndType(slide.getUri(), slide.getContentType());
        context.startActivity(intent);
    }
}
```

Listing 4.2: Vulnerable code 1 for Rule 1.

```
// rest of the code
Intent installIntent = new Intent(Intent.ACTION_VIEW);
installIntent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
installIntent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
installIntent.putExtra(Intent.EXTRA_NOT_UNKNOWN_SOURCE, true);
installIntent.setDataAndType(fileUri, downloadManager.
    getMimeTypeForDownloadedFile(referenceId));
try  {
    Log.d(TAG, "onDownloadComplete startActivity(installIntent)");
    startActivity(installIntent);
} catch(Exception ex)  {
    Log.d(TAG, "# onDownloadComplete startActivity(installIntent) ex="+ex);
    Toast.makeText(activity,"APK downloaded. Cannot offer to install.", Toast.
    LENGTH_LONG).show();
} // rest of the code
```

Listing 4.3: Vulnerable code 2 for Rule 1.

As it is possible to notice, the two vulnerable snippets may be different at structure level, but the suggested fix for both codes results to be very similar:

```java
private void previewImageDraft(final @NonNull Slide slide)  {
    if (MediaPreviewActivity.isContentTypeSupported(slide.getContentType()) &&
    slide.getUri() != null)  {
        Intent intent = new Intent(context, MediaPreviewActivity.class);
        intent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
        intent.putExtra(MediaPreviewActivity.SIZE_EXTRA, slide.asAttachment().
    getSize());
        intent.setDataAndType(slide.getUri(), slide.getContentType());
        // fix1
        List<ResolveInfo> possibleActivitiesList = getPackageManager().
    queryIntentActivities(intent, PackageManager.MATCH_ALL);
        if (possibleActivitiesList.size() > 1) {
            String title = getResources().getString(R.string.chooser_title);
            Intent chooser = Intent.createChooser(intent, title);
            startActivity(chooser);
        } else {  context.startActivity(intent); }
    }
}
```

Listing 4.4: Ground truth for vulnerable code 1.

```java
// rest of the code
Intent installIntent = new Intent(Intent.ACTION_VIEW);
installIntent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
installIntent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
installIntent.putExtra(Intent.EXTRA_NOT_UNKNOWN_SOURCE, true);
installIntent.setDataAndType(fileUri, downloadManager.
    getMimeTypeForDownloadedFile(referenceId));
try  {
    Log.d(TAG, "onDownloadComplete startActivity(installIntent)");
    // fix1
    List<ResolveInfo> possibleActivitiesList = getPackageManager().
    queryIntentActivities(installIntent, PackageManager.MATCH_ALL);
    if (possibleActivitiesList.size() > 1) {
        String title = getResources().getString(R.string.chooser_title);
        Intent chooser = Intent.createChooser(installIntent, title);
        startActivity(chooser);
    } else { startActivity(installIntent); } } // rest of the code
```

Listing 4.5: Ground truth for vulnerable code 2.

The best practice associated with Rule 1 is the following: *"Show an app chooser. Implicit intents may be handled by more than one activity, therefore show an app chooser before launching the intent."*. As a result, the common introduced fix to let the snippet be compliant with the guideline is the one showed in

```java
// fix1
List<ResolveInfo> possibleActivitiesList = getPackageManager().
    queryIntentActivities(<intent_name>, PackageManager.MATCH_ALL);
if (possibleActivitiesList.size() > 1) {
    String title = getResources().getString(R.string.chooser_title);
    Intent chooser = Intent.createChooser(<intent_name>, title);
    startActivity(chooser);
}
//
```

Listing 4.6: Ground truth for vulnerable code 2.

The above fix has been introduced in Listing 4.2 and Listing 4.3, respectively at line 7 and 9. The given example wants to suggest how structured the manually generated fixes are, in order to be correctly introduced in the general structure of the vulnerable snippet.

Maintaining a coherence among the implementation of fixes from the same category allowed to reach a common structure, which permits to affirm that if a fix is integrated in a compilable application and does not introduce any compilation error, then with very high probabilities the patch proposed for a different vulnerability from the same category, will also compile. This is necessary to demonstrate what it has been stated in Section 4.2 and that every manually generated fix which serves as ground truth, are very likely not to introduce compilation errors when applied to the original application. The same is done for vulnerabilities regarding the Android Manifest file.

4.7 FIXES EVALUATION

Fixes proposed serve as a ground truth for evaluating the abilities of LLMs in suggesting patches for the same vulnerability in input. Since most fixes require final decisions from developers of the original application, it is not possible to absolutely guarantee that the integration of the patch will preserve the logic of the application. Furthermore, there are fixes which necessarily will lead to a change in the application's behaviour due to the origin of a fix itself: modify the current behaviour to pursue the implementation of best practice. On the other hand, it is possible to guarantee that proposed repairs do not violate other rules (best practices) and that they fix the original source code.

Once the repair is done, SPECK has been used to check whether the previously detected vulnerability is still present or not. The modified application is compiled into APK and analysed by SPECK but, as stated in Section 4.5, not every application in the considered analysis can be compiled, due to dependency problems or not having the needed files. However, in light of the structured format of fixes, it is possible to state that: if a repair for a certain vulnerability category is integrated into an application and such an application compiles, then it will likely let every other application requiring the same patch, compile. Hence, for each type of proposed repair, at least one application that already comes with no compilation problems, is tested against SPECK after the integration of the fix. In case of fix types for which none of the corresponding vulnerabilities has been extracted from an application that can successfully compile, then a random application is chosen and a crafted project is created to recreate it, which now can be compiled and the fix under analysis being tested. The crafted project includes the exact vulnerability and recreates all the necessary information and classes of the original source code, to be as close as possible to the true application. In addition, for fixes that require the involvement of specific values as variable, abstract values have been assigned just to assert that the fix does not introduce any error.

By doing so, every fix type has been tested, assuring that if introduced in the original application, this can still be compiled without any new problem and, more importantly, the previously detected vulnerability is no more present.

5

LLMs Evaluation

The collection of Android applications, the realisation of a new dataset of vulnerabilities specifically of the Android OS can be certainly a contribution to the research community in patching vulnerabilities, but it can also be considered as a prerequisite in order to perform LLMs evaluation on Android vulnerabilities from real-word applications, which is the core objective of this work.

The advancement of new large language models and the improvements of the already existing ones is pushing every research field to explore them, trying to study how far they can go in knowledge and, consequently, how good they are in ongoing research problems. Among them, the APR research community has also shifted its direction towards the world of pre-trained LLMs, to understand how well they can perform in common APR tasks, such as code refinement and the related task of defect detection. The power of LLMs lies in their ability to learn complex relationships among data and, thanks to the very large amount of data they are pre-trained on, LLMs result in being good even when facing new problems. To better say, given an APR problem, a general purpose LLM can outperform traditional APR tools which were born with the specific objective of being tools for repairing software defects. As a consequence, fine-tuned LLMs can achieve higher accuracy in results and can only lead LLMs to be the best option in solving such problems.

5.1 ChatGPT, Bard and Studio Bot

For the sake of this project, three LLMs have been evaluated: ChatGPT-3.5, Google Bard and Android Studio Bot. The last one, as stated by Google [40], is different from other LLM-powered chatbots since "Studio Bot is tightly integrated within Android Studio, which means it can provide more relevant responses" and "Studio Bot leverages an LLM that was designed to help with coding scenarios". Hence, Android Studio Bot is supposed to be the most accurate one in suggesting patches for Android vulnerabilities.

The reason why these three LLMs have been chosen to be evaluated is explained as follows. Considering a developer working on an Android application, dealing with compilation errors, bugs and doubts, it is very likely that he or she will choose an interactive LLM, free to use and already well implemented. In case of vulnerabilities, they are involuntary most of the time, unconsciously introduced by the developers themselves; then, they would not even use an LLM since they are unaware of the vulnerability. Now, suppose there is a supported detection tool – SPECK, in this case – which is able to detect whether a best practice is implemented or not and therefore if there is a possible vulnerability, then the developer will use an interactive LLM to know how to implement such best practice with very high probability. Thus, imagining such a scenario, the choice of LLMs came naturally. ChatGPT and Bard are the most used interactive LLMs, extremely easy to use and make the interaction seem like chatting. In addition, Android Studio Bot is analysed, since it not only has the interactive aspective but it is also strictly related to Android.

Undoubtedly, in a different scenario where there is the actual need to integrate the abilities of the LLMs in a wider context, then it would make sense to consider also other LLMs, working at APIs level and fine-tuning them to reach better results for the specific problem and scope.

5.2 Prompt Engineering

Prompt engineering is a new research field born along with the advancement of LLMs, since their outputs resulted to be strictly correlated with the type of input (prompt) they receive. A prompt can be defined as the starting point of the interaction with a large language model and it is usually a natural description of the task the LLM has to accomplish. Commonly, prompts are divided into three main categories: basic prompt, in-context prompt, task-specific prompt [41]. In particular:

1. a **basic prompt** contains a general definition – the natural description of the task – of the problem and the expected output, sometimes with an input code;

2. **in-context prompt** includes one or more examples of input and expected output, within the basic prompt;

3. **task-specific prompt**: a different prompt is defined for each task type.

Basic prompt can be considered as a zero-shot learning scenario, where the LLM is not learning anything new from the prompt, but it just returns an output based on the already integrated knowledge. The *in-context* prompt involves a sort of in-context learning, where the prompt also contains examples of input-output pairs which should guide the LLM to return a more accurate output. This is a type of few-shot learning, whose goal is the same as fine-tuning: both strategies aim to let the LLM become "clever" with respect to the specific task and provide improved answers to the inputs; differences lie in how such an objective is reached.

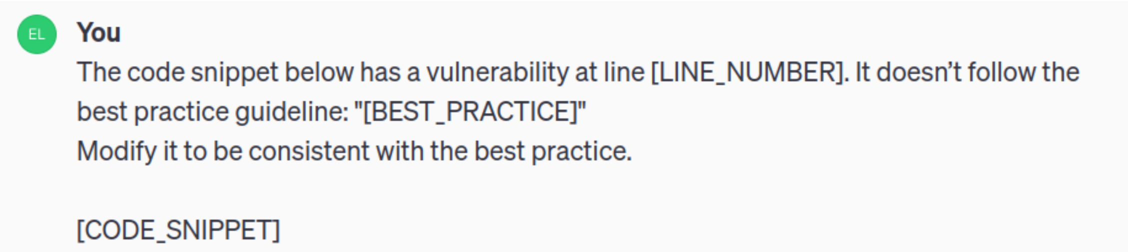
Regarding the goal of this project, the basic prompt is the chosen one to evaluate LLMs. The rationale behind this choice is given by several motivations: firstly, considering once again the scenario described above (Section 5.1) and the developer who is likely to ask his or her problems to an interactive LLM, such LLMs are also general-purpose; consequently, the most common scenario involves the case of a basic-prompt. Subsequently, in-context learning would require a bunch of examples including both input and expected output. Taking into consideration the challenges and limitations faced in gathering a relevant amount of vulnerabilities for the dataset, it would have been quite challenging to find other examples to give as input. However, in-context learning turned out not to be necessary, considering the final results described in detail in Chapter 7.

Finally, a *task-specific prompt* is not suitable for this project since there is one main task to be performed: vulnerability repair. Vulnerabilities found and analysed in this study are yet divided in categories, but they still fall into the single domain of Android vulnerabilities.

5.2.1 THE PROMPT

This subsection is dedicated to a brief description of the prompt used to test and then evaluate the LLMs. The way LLMs performed is influenced by the type of input they receive: a "poor" prompt can mislead the model to a different task and result in bad performance, while an extremely detailed prompt can favour to obtain the desired output.

For this project, the evaluation of LLMs regards how accurate and correct the generated fix is, considering the vulnerable code snippet given as input. Since vulnerability detection performed by LLMs is not an objective of this study, the vulnerable line is included in the prompt along with the best practices that the line – or the whole snippet – violates. The template used to generate the specific prompt is the following one:



EL

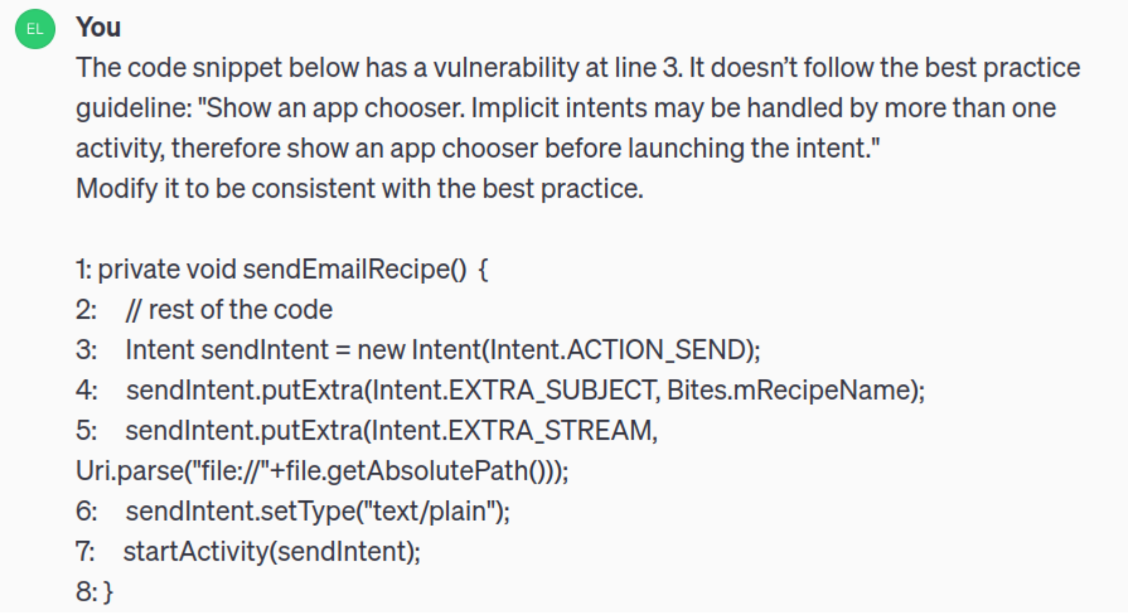
You

The code snippet below has a vulnerability at line [LINE_NUMBER]. It doesn't follow the best practice guideline: "[BEST_PRACTICE]"
Modify it to be consistent with the best practice.

[CODE_SNIPPET]

Figure 5.1: Template for a basic prompt.

which turns out to be like the example below, when considering a specific vulnerability:



EL

You

The code snippet below has a vulnerability at line 3. It doesn't follow the best practice guideline: "Show an app chooser. Implicit intents may be handled by more than one activity, therefore show an app chooser before launching the intent."
Modify it to be consistent with the best practice.

```
1: private void sendEmailRecipe() {
2:    // rest of the code
3:    Intent sendIntent = new Intent(Intent.ACTION_SEND);
4:    sendIntent.putExtra(Intent.EXTRA_SUBJECT, Bites.mRecipeName);
5:    sendIntent.putExtra(Intent.EXTRA_STREAM,
Uri.parse("file://"+file.getAbsolutePath()));
6:    sendIntent.setType("text/plain");
7:    startActivity(sendIntent);
8: }
```

Figure 5.2: Example of a basic prompt.

As it is possible to understand, the chosen basic-prompt includes a short description for the best practice that wants to be as clear as possible and directly pointing out the problem.

Once having generated complete prompts for each vulnerable code snippet, all the three LLMs (ChatGPT, Bard and Studio Bot) have been tested with the same prompt for each vulnerability. Using the same input for all the models allows an objective evaluation: it permits a parallel analysis of three different suggested repair (one for each LLM) against the reference fixed code.

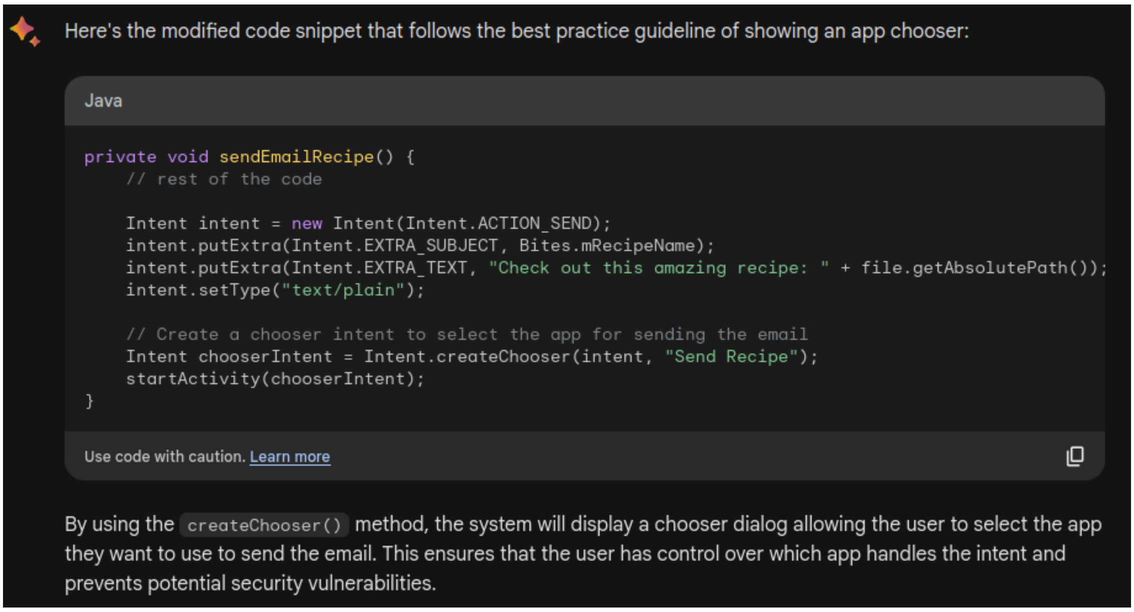
For every new prompt, a new chat have been started, to avoid any involuntary learning and ensure that each output is obtained in a *zero-shot* scenario. Moreover, looking at the output of Studio Bot (Figure 5.5), it is necessary to underlying that the generated code is written in Kotlin, regardless of the code in input which is written in Java. Showing this inability of directly understanding the language of the input code, its prompts have been modified just by adding the specification "*Use Java*", to allow a parallel comparison between the three LLMs.

Each prompt is, of course, given as input only once and the corresponding output is directly taken as valid to be pre-processed and analysed, regardless of the possibility of having a different output which may be correct or not, with respect to the previous one. This is due to the non-deterministic nature of the models, since they are not providing always the same output for the same prompt given as input multiple times. A more detailed analysis on this aspect may be considered as a future work, as stated in Chapter 8.

5.3 LLMs Outputs

How well LLMs performed on generating patches for all the vulnerabilities in the dataset will be described in Chapter 7, which gathers all the results obtained.

Generally speaking, outputs of all the three LLMs chosen for this analysis are a mix of code and literal descriptions. For the objective of this project, only the generated code is considered. Figures below show a typical output for Bard (Figure 5.3), ChatGPT (Figure 5.4) and Studio Bot (Figure 5.5) which have been prompted with the prompt in Figure 5.2.



Here's the modified code snippet that follows the best practice guideline of showing an app chooser:

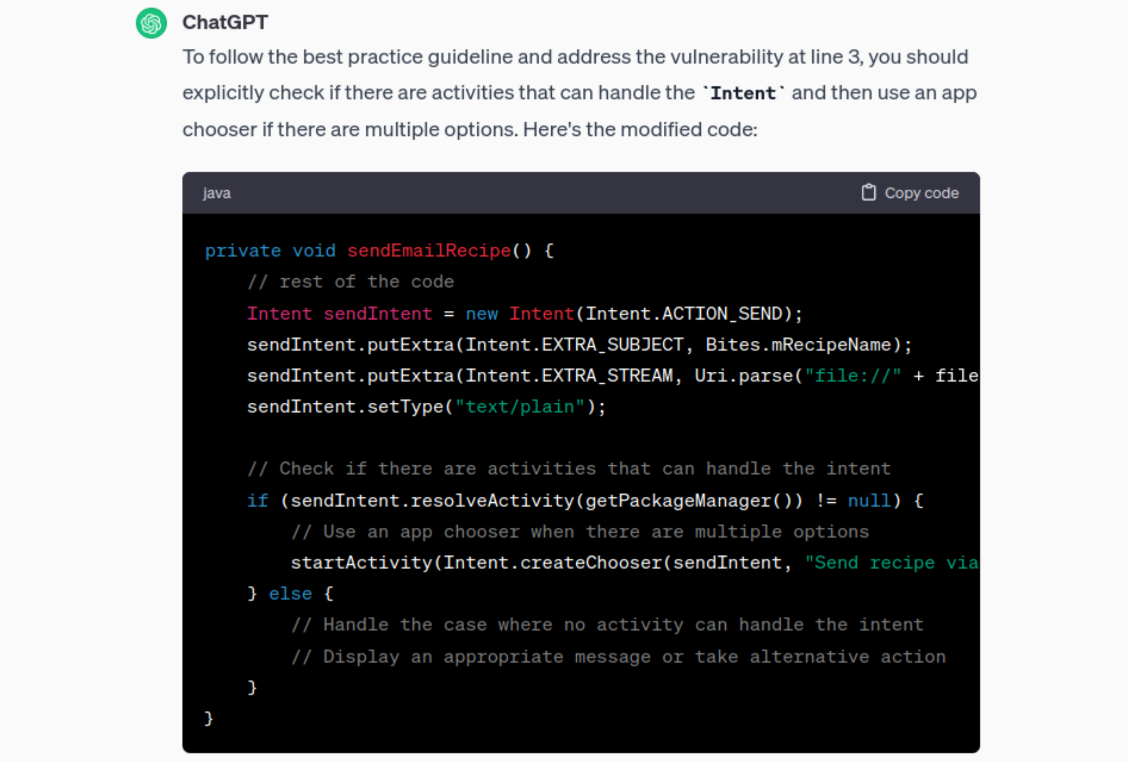
```java
private void sendEmailRecipe() {
    // rest of the code

    Intent intent = new Intent(Intent.ACTION_SEND);
    intent.putExtra(Intent.EXTRA_SUBJECT, Bites.mRecipeName);
    intent.putExtra(Intent.EXTRA_TEXT, "Check out this amazing recipe: " + file.getAbsolutePath());
    intent.setType("text/plain");

    // Create a chooser intent to select the app for sending the email
    Intent chooserIntent = Intent.createChooser(intent, "Send Recipe");
    startActivity(chooserIntent);
}
```

By using the `createChooser()` method, the system will display a chooser dialog allowing the user to select the app they want to use to send the email. This ensures that the user has control over which app handles the intent and prevents potential security vulnerabilities.

Figure 5.3: Typical output for Bard.

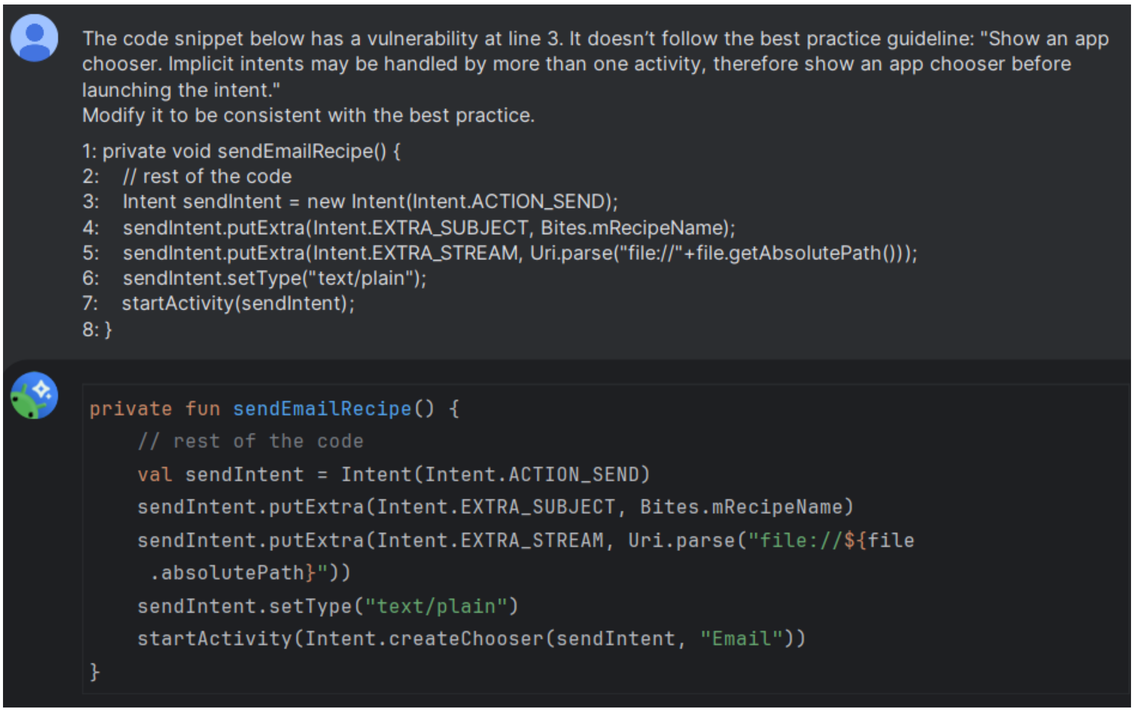


To follow the best practice guideline and address the vulnerability at line 3, you should explicitly check if there are activities that can handle the `Intent` and then use an app chooser if there are multiple options. Here's the modified code:

```java
private void sendEmailRecipe() {
    // rest of the code
    Intent sendIntent = new Intent(Intent.ACTION_SEND);
    sendIntent.putExtra(Intent.EXTRA_SUBJECT, Bites.mRecipeName);
    sendIntent.putExtra(Intent.EXTRA_STREAM, Uri.parse("file://" + file
    sendIntent.setType("text/plain");

    // Check if there are activities that can handle the intent
    if (sendIntent.resolveActivity(getPackageManager()) != null) {
        // Use an app chooser when there are multiple options
        startActivity(Intent.createChooser(sendIntent, "Send recipe via
    } else {
        // Handle the case where no activity can handle the intent
        // Display an appropriate message or take alternative action
    }
}
```

Figure 5.4: Typical output for ChatGPT.



The code snippet below has a vulnerability at line 3. It doesn't follow the best practice guideline: "Show an app chooser. Implicit intents may be handled by more than one activity, therefore show an app chooser before launching the intent."
Modify it to be consistent with the best practice.

```
1: private void sendEmailRecipe() {
2:    // rest of the code
3:    Intent sendIntent = new Intent(Intent.ACTION_SEND);
4:    sendIntent.putExtra(Intent.EXTRA_SUBJECT, Bites.mRecipeName);
5:    sendIntent.putExtra(Intent.EXTRA_STREAM, Uri.parse("file://"+file.getAbsolutePath()));
6:    sendIntent.setType("text/plain");
7:    startActivity(sendIntent);
8: }
```

```
private fun sendEmailRecipe() {
    // rest of the code
    val sendIntent = Intent(Intent.ACTION_SEND)
    sendIntent.putExtra(Intent.EXTRA_SUBJECT, Bites.mRecipeName)
    sendIntent.putExtra(Intent.EXTRA_STREAM, Uri.parse("file://${file
     .absolutePath}"))
    sendIntent.setType("text/plain")
    startActivity(Intent.createChooser(sendIntent, "Email"))
}
```

Figure 5.5: Typical output for Studio Bot.

5.4 PRE-PROCESSING PHASE

As it is possible to notice from previous section, even if only the proposed code is extracted, it may also include comments to guide through the readability of the snippet itself. Consequently, since the final goal is to evaluate the code they generated and understand if it is a correct fix for the input vulnerability, all outputs have been pre-processed before being compared with respective ground truth. Naturally, the same pre-processing is applied to both prediction (LLM output) and reference (ground truth) for consistency reasons.

This phase is also necessary in order to work with chosen metrics for evaluation, which are described in Chapter 6.

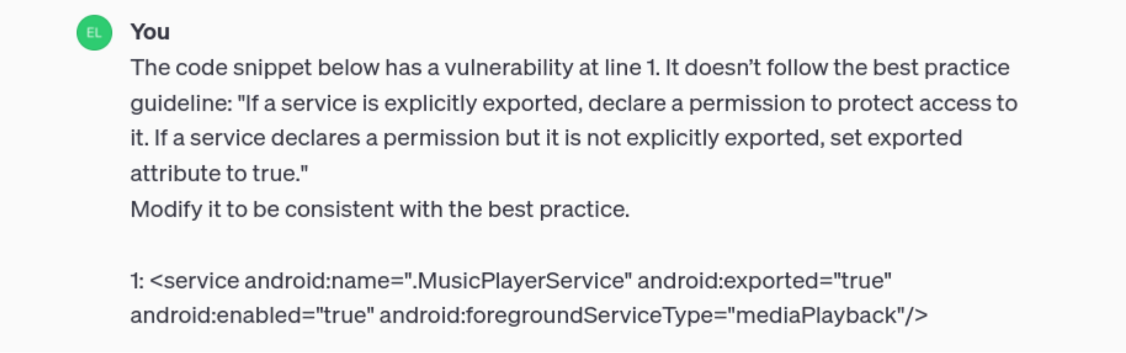
5.4.1 PRE-PROCESSING JAVA-RELATED OUTPUTS

To extract only the relevant information, which is indeed the generated code, all outputs have been cleaned from comments that came along with the code. For outputs related to prompts with Java vulnerability, also additional methods which do not serve the purposes of the evaluation (e.g. the main method) have been removed. In addition, some of these generated codes resulted in not having balanced brackets and this can lead to parsing errors and the unavailability of the code when it comes to be compared with the ground truth, resulting in an undefined similarity score (see Section 6.3).

It is worth noting that the objective of this evaluation is to understand whether the proposed code either repairs or not an input vulnerability, without taking into account syntax errors, which fall outside the scope of this project. For this reason, once cleaned from comments and additional methods, all the outputs have been parsed and unmatched brackets have been manually fixed; naturally, the proposed code has not been modified at all. After having pre-processed all the outputs from the three LLMs, each suggested repair is compared with the corresponding ground truth.

5.4.2 PRE-PROCESSING XML-RELATED OUTPUTS

In case of outputs related to the vulnerabilities extracted from the Android Manifest file, pre-processing revealed to be fundamental and significant. The first step is the same of the pre-processing phase for Java suggested repairs, which is the removal of natural sentences and comments, to extract only the relevant snippet of code. To be as clear as possible, an example of XML-related prompt, respective output (LLM proposed fix, Listing 5.1) and corresponding ground (Listing 5.2) are shown as follows:



You

The code snippet below has a vulnerability at line 1. It doesn’t follow the best practice guideline: "If a service is explicitly exported, declare a permission to protect access to it. If a service declares a permission but it is not explicitly exported, set exported attribute to true."

Modify it to be consistent with the best practice.

```
1: <service android:name=".MusicPlayerService" android:exported="true"
android:enabled="true" android:foregroundServiceType="mediaPlayback"/>
```

Figure 5.6: Prompt to ChatGPT for XML-related vulnerability.

```xml
<service android:name=".MusicPlayerService" android:exported="true"
    android:enabled="true" android:foregroundServiceType="mediaPlayback">
    <intent-filter>
        <action android:name="com.example.YOUR_ACTION" />
    </intent-filter>
    <permission android:name="com.example.YOUR_PERMISSION" />
</service>
```

Listing 5.1: Proposed fix for Android Manifest vulnerability by ChatGPT.

```xml
<permission
    android:name="custom_permission"
    android:protectionLevel="normal"/>
<service
    android:name=".MusicPlayerService"
    android:exported="true" android:enabled="true"
    android:foregroundServiceType="mediaPlayback"
    android:permission="custom_permission" />
```

Listing 5.2: Ground truth for vulnerable code in prompt in Figure 4.6

Since the metrics used for the comparison with ground truth are commonly used for text similarity (see Section 6.4), a first type of pre-processing can include the removal of special characters and transforming the snippet into a sentence where all the words are concatenated (having whitespaces as delimiter). The pre-processing is a ”simple” word-tokenizer and it will be referenced as ”*pre-processing type 1*”. The reason why this type of metrics and pre-processing are considered is due to the structured format of the XML snippets, whose structure is more consistent than a typical Java code.

However, this pre-processing can produce misleading results. Most outputs from Bard and ChatGPT include additional tags in the proposed repair, which are not actually requested for the aim of the fix. To provide a better understanding, with this pre-process the codes in Listing 5.1 and Listing 5.2 are respectively turned in:

```
service android name .MusicPlayerService android exported true android enabled
    true android foregroundServiceType mediaPlayback intentfilter action
    android name com.example.YOURACTION intentfilter permission android name
    com.example.YOURPERMISSION service
```

```
permission android name custompermission android protectionLevel normal service
     android name .MusicPlayerService android exported true android enabled
    true android foregroundServiceType mediaPlayback android permission
    custompermission
```

Once again, since the final goal is to understand whether the output is correct or not in terms of fixing the specific vulnerability given as input, a different pre-processing is also proposed and tested. Since metrics for text similarity are involved, differences in the snippets lengths or in words composing the snippets are a crucial point when computing the similarity score. Hence, another type of pre-processing is proposed and it will be referred as ”*pre-processing type 2*”. It sees the additional tags removed and it includes the abstraction for attribute values which are commonly arbitrary. For instance, the `android:permission` attribute value can be whatever, depending on the permission to assign (e.g. custom permission name is chosen by the developer). Moreover, the whole snippet is parsed using an XML parser and is formatted, to allow a consistent comparison between it and its respective ground truth; for instance, if the component is a service and it has attributes such as `android:name`, then it becomes `service.name`. Using *pre-processing type 2*, the codes in Listing 5.1 and Listing 5.2 are respectively turned in:

```
service.name .MusicPlayerService service.exported true service.enabled true service.foregroundServiceType mediaPlayback
```

```
permission.name custom_permission permission.protectionLevel normal service.name .MusicPlayerService service.exported true service.enabled true service.foregroundServiceType mediaPlayback service.permission custom_permission
```

In the first case, new tags added by the LLM which are not included in the original prompt, are removed; consequently, the output of the LLM when pre-processed does not show the `<intent-filter>` tag anymore. In addition, since the `<permission>` tag is not placed as it is supposed to be (outside the declaration of the component), it considered as missing and, therefore, not included in the final pre-processed output.

In conclusion, the comparison becomes more reliable than if performed with two snippets pre-processed using *pre-processing type 1*.

6

Results

This Chapter gathers all the results of this evaluation. In order to understand them, all the used metrics are described the following section.

6.1 METRICS

In general, the task of producing a patch starting from a vulnerable code can be usually treated as a code generation task which, in this case, can be referred to as code refinement task whose aim is exactly to automatically fix general bugs in the code. For the scenario under analysis, models know in advance where the vulnerability is and the natural description of what to do to produce a patch.

One of the most used metric also in learning-based techniques for APR is BLEU (Bilingual Evaluation Understudy), even if it is commonly used for comparing natural language sentences. In APR field, it is also widely used and measures the similarity between the generated output (from a Large Language Model) and the corresponding reference, usually made by humans. However, it is not the best option when it comes to compare two code snippets, since there are several aspects that can mislead its output:

- different variable or methods names in references and output, due to the fact that LLMs can introduce new variables or adding methods necessary to produce the patch;

- BLEU only compares the sequence of words, which cannot be considered as an accurate approach to check similarity between codes.

To catch also semantic information from a snippet of code and to base the comparison also on this aspect and not only on similar sequence of words, only CodeBLEU is used. Specifically speaking, it compares the similarity between generated patches and relative references for vulnerabilities extracted from Java classes. It automatically parses the Java code and computes the final score by evaluating four different aspects:

- **n_gram match**: an n-gram is a sequence of n words; the match is given by the comparison of the n-grams of the two inputs sequences;
- **weighted_n_gram match**: given n-grams, they have a weight associated with them which allow to focus on relevant aspect of the code as input. The match is given by computing the similarity between weighted n-grams;
- **syntactic_match**: after parsing the two input codes, their syntax is extracted and the structure of the two codes (their Abstract Syntax Tree) is compared without taking into account semantic similarity;
- **dataflow_match**: when parsing the codes, the flow of data is extracted. The input codes are compared on the basis of how data flows in them: if the structure is different, but the data follows the same or a similar path, then there is match in terms of code behaviour.

These values comes together in the CodeBLEU score as stated in the following equation:

$$\begin{aligned} CodeBLEU = \alpha \times n\_gram\_match + \beta \times weighted\_n\_gram\_match \\ + \gamma \times syntactic\_match + \delta \times dataflow\_match \end{aligned} \tag{6.1}$$

Modifying the weights associated to the different scores, it is possible to notice which of them results in being more relevant a the specific task; the goal of the evaluation is to measure how close the generated patches are to the corresponding reference in terms of semantic similarity. Hence, in addition to the computation of CodeBLEU where all the components have the same weight, other computations are performed to consider the syntactic and dataflow match more than the other two. In the matter of vulnerabilities extracted from the Android Manifest file which are therefore written using xml, other metrics are considered, which are commonly used for comparing texts and compute their similarity. They include:

- **Jaccard Similarity**: also known as "intersection over union". The two sentences in inputs are considered as a set of words and the similarity score is given by the total number of common words between the two sentences over the total number of words. The score is a value between 0 (completely different) and 1 (identical);
- **Cosine Similarity**: texts in inputs are transformed in vectors and the cosine of the related angle indicates how close this two vectors are; it ranges between 0 (completely different) and 1 (identical);
- **Levenshtein Distance**: it computes the total number of modifications at character-level (insertion, deletion, substitution) needed to transform one sentence into the other; the score is a value between 0 (identical) and 1 (completely different).

For consistency reasons, the actual value considered from Levenshtein Distance is $1 - lev(a, b)$, to have all the three metrics following the same criteria. These metrics provide a quantitative analysis of the number of plausible or implausible patches, where the plausible patch is intended to be the one similar to the respective reference patch; the implausible one is considered to be consistently different from ground truth. Naturally, human validation is conducted for each generated patch of every LLM under analysis, to assure a semantic equivalence and therefore, the correctness of the patch.

6.2 NOTES ABOUT RESULTS TABLES

To be as clear as possible, results are presented by separating the Java vulnerabilities from the XML-related ones, also because the metrics used for the two categories are different. At first, LLMs are going to be evaluated for their performances on suggesting repairs for Java and for XML vulnerabilities, separately. At a later stage, general considerations will be explained.

Now, let's suppose that the original dataset has been divided in two, one for Java-related vulnerabilities and one for XML-related ones. All the tables report the similarity score between ground truths and proposed fixes by the language model considered, for a specific class of vulnerabilities (either from Java or XML). In particular, given a vulnerability at index i in the considered original dataset (either from Java or XML), then at index i in each one of these tables there is the similarity score computed between corresponding ground truth and relative suggested fix, for the same vulnerability at i.

6.3 CodeBLEU for Java-related comparison

6.3.1 CodeBLEU for ChatGPT-3.5 Outputs

Table 6.1 reports the single scores needed to compute CodeBLEU.

index	n_gram_match	weighted_n_gram_match	syntactic_match	dataflow_match
1	0.6122643512	0.6301058114	0.6785714286	0.8
2	0.5943198196	0.6034361302	0.7	0.8
3	0.7261813303	0.7313136731	0.8709677419	1
4	0.3934549377	0.6238986072	0.3333333333	0.6666666667
...	...	...	...	...
185	0.6177986464	0.6051414259	0.5555555556	0.6
186	0.6106491161	0.813061341	0.9038461538	1
187	0.3158397558	0.5311122703	0.8035714286	0.972972973
188	0.7016879391	0.7063197941	1	0.8

Table 6.1: Metrics for evaluating CodeBLEU for repairs from ChatGPT-3.5 for Java-related vulnerabilities.

Table 6.2 reports the actual scores of three CodeBLEU configurations (described in Section 7.1).

index	codeBLEU_1	codeBLEU_2	codeBLEU_3
1	0.6802353978	0.7038555244	0.7083957451
2	0.67744389874	0.7046633925	0.7052910145
3	0.8321156863	0.8734629602	0.8731884868
4	0.5043383862	0.5026030317	0.5453201456
...	...	...	...
185	0.594623907	0.5878854553	0.59195411
186	0.8318891527	0.8799027224	0.8902917853
187	0.6558741069	0.74888333445	0.7544440895
188	0.8020019333	0.84120116	0.8167487424

Table 6.2: Corresponding CodeBLEU scores from 6.1 for ChatGPT-3.5 for Java-related vulnerabilities.

6.3.2 CodeBLEU for Bard Outputs

Table 6.4 reports the single scores needed to compute CodeBLEU.

index	n_gram_match	weighted_n_gram_match	syntactic_match	dataflow_match
1	0.6122643512	0.6301058114	0.6785714286	0.8
2	0.005516564421	0.09913646904	0.2333333333	0.2
3	0.2582326872	0.6099792108	0.8064516129	0.7142857143
4	0.1025673262	0.5219910253	0	0.6666666667
...	...	...	...	...
185	0.610714568	0.8264569367	0.7777777778	0.6
186	0.7799306703	0.813061341	0.8846153846	1
187	0.7518285148	0.7747623579	0.8392857143	0.8918918919
188	0.5954165059	0.5962483963	0.95	0.7

Table 6.3: Metrics for evaluating CodeBLEU for repairs from Google Bard for Java-related vulnerabilities.

Table 6.4 reports the actual scores of three CodeBLEU configurations (described in Section 7.1).

index	codeBLEU_1	codeBLEU_2	codeBLEU_3
1	0.6802353978	0.7038555244	0.7083957451
2	0.1344965917	0.1673646217	0.163669107
3	0.5972373063	0.6624898492	0.6656452604
4	0.3228062545	0.3270170861	0.4074211556
...	...	...	...
185	0.7037373206	0.6977979479	0.7021301354
186	0.869401849	0.8985641863	0.9024122484
187	0.8144421197	0.8349007931	0.8354516263
188	0.7104162256	0.7562497353	0.7261037497

Table 6.4: Corresponding CodeBLEU scores from 6.3 for Google Bard for Java-related vulnerabilities.

6.3.3 CodeBLEU for Android Studio Bot Outputs

Table 6.5 reports the single scores needed to compute CodeBLEU.

index	n_gram_match	weighted_n_gram_match	syntactic_match	dataflow_match
1	0.6122643512	0.6301058114	0.6785714286	0.8
2	0.6122643512	0.6301058114	0.7	0.8
3	0.7872223756	0.7899291974	0.8709677419	1
4	0.3573352482	0.7734940377	0.3333333333	0.6666666667
...	...	...	...	...
185	0.8115699932	0.8221184302	0.75	0.6
186	0.5978251158	0.6696373429	0.7884615385	0.8181818182
187	0.7518285148	0.7747623579	0.8392857143	0.8918918919
188	0.4779995354	0.4805622828	0.75	0.3

Table 6.5: Metrics for evaluating CodeBLEU for repairs from Android Studio Bot for Java-related vulnerabilities.

Table 6.6 reports the actual scores of three CodeBLEU configurations (described in Section 7.1).

index	codeBLEU_1	codeBLEU_2	codeBLEU_3
1	0.6802353978	0.7038555244	0.7083957451
2	0.6855925407	0.7113555244	0.713752888
3	0.8620298287	0.8914114456	0.8939464724
4	0.5327073215	0.5196243929	0.5791070343
...	...	...	...
185	0.7459221059	0.7175532635	0.7141866069
186	0.7185264538	0.7524445436	0.7515799592
187	0.8144421197	0.8349007931	0.8354516263
188	0.5021404546	0.5112842727	0.4754405242

Table 6.6: Corresponding CodeBLEU scores from 6.5 for Google Bard for Java-related vulnerabilities.

6.4 METRICS FOR XML-RELATED COMPARISON

This section reports the scores for each metric used to compare XML-related outputs with respective ground truth after pre-processing both of them (see Subsection 5.4.2).

6.4.1 METRICS FOR CHATGPT-3.5 OUTPUTS

Table 6.7 reports the scores for similarity after *pre-processing type 1*.

index	jaccard_1	levenshtein_1	cosine_1
1	1	0.98125	1
2	0.4	0.4691943128	0.7644707872
...	...	...	...
14	0.7	0.5885167464	0.9144009144
15	0.7	0.5625	0.9296696802
16	0.4666666667	0.3648208469	0.8620601237
...	...	...	...
95	0.9285714286	0.4157608696	0.8967442657
96	0.4210526316	0.4596577017	0.7980986476

Table 6.7: Metrics for evaluating repairs from ChatGPT-3.5 for XML-related vulnerabilities (with pre-process 1).

Table 6.8 reports the scores for similarity after *pre-processing type 2*.

index	jaccard_2	levenshtein_2	cosine_2
1	1	1	1
2	1	1	1
...	...	...	...
14	0.6666666667	0.6616915423	0.905807819
15	0.6666666667	0.6324324324	0.8892972918
16	0.5	0.4	0.785905248
...	...	...	...
95	1	0.4981549815	1
96	0.7	0.7745098039	0.9356014857

Table 6.8: Metrics for evaluating repairs from ChatGPT-3.5 for XML-related vulnerabilities (with pre-process 2).

6.4.2 METRICS FOR GOOGLE BARD OUTPUTS

Table 6.9 reports the scores for similarity after *pre-processing type 1*.

index	jaccard_1	levenshtein_1	cosine_1
1	1	0.68125	1
2	1	0.9292929293	1
...	...	...	...
14	0.7	0.5980861244	0.9253627917
15	0.7	0.5052083333	0.9486832981
16	0.8888888889	0.7567567568	0.9360565811
...	...	...	...
95	0.7647058824	0.385	0.9289853059
96	0.7	0.6	0.9216805029

Table 6.9: Metrics for evaluating repairs from Google Bard for XML-related vulnerabilities (with pre-process 1).

Table 6.10 reports the scores for similarity after *pre-processing type 2*.

index	jaccard_2	levenshtein_2	cosine_2
1	1	0.5869565217	1
2	1	1	1
...	...	...	...
14	0.6666666667	0.6616915423	0.905807819
15	0.6666666667	0.6324324324	0.8892972918
16	0.7	0.6114285714	0.907841299
...	...	...	...
95	0.6153846154	0.6088560886	0.8436958339
96	0.6666666667	0.6666666667	0.8892972918

Table 6.10: Metrics for evaluating repairs from Google Bard for XML-related vulnerabilities (with pre-process 2).

6.4.3 METRICS FOR ANDROID STUDIO BOT OUTPUTS

Table 6.11 reports the scores for similarity after *pre-processing type 1*.

index	jaccard_1	levenshtein_1	cosine_1
1	1	0.68125	1
2	1	0.7575757576	0.9594032236
...	...	...	...
14	0.3636363636	0.3732057416	0.8164965809
15	0.5	0.5416666667	0.8082903769
16	0.8	0.6687116564	0.8805874421
...	...	...	...
95	0.7857142857	0.4571428571	0.9324566031
96	0.5454545455	0.4761904762	0.9223309842

Table 6.11: Metrics for evaluating repairs from Android Studio Bot for XML-related vulnerabilities (with pre-process 1).

Table 6.12 reports the scores for similarity after *pre-processing type 2*.

index	jaccard_2	levenshtein_2	cosine_2
1	1	0.5869565217	1
2	1	1	0
...	...	...	...
14	0.1818181818	0.2537313433	0.6625891564
15	0.3	0.4162162162	0.672213997
16	0.6363636364	0.5885714286	0.8819171037
...	...	...	...
95	0.7692307692	0.4686346863	0.9296696802
96	0.5	0.6470588235	0.8488746876

Table 6.12: Metrics for evaluating repairs from Android Studio Bot for XML-related vulnerabilities (with pre-process 2).

6.5 HUMAN VALIDATION

This section provides the number of correct (True Positive) and incorrect (True Negative), computed by manually validating one by one the patches proposed by the single LLMs, to state if they are semantically equivalent (or not) to the relative ground truth.

	# TP	# TN	# total	% TP	% TN	% total
ChatGPT-3.5	240	44	284	84.5%	15.5%	100%
Google Bard	254	30	284	89.4%	10.6%	100%
Android Studio Bot	96	188	284	33.8%	66.2%	100%

Table 6.13: Number of True Positive (TP) and True Negative (TN) comparing LLMs outputs to ground truth.

6.5.1 JAVA-RELATED OUTPUTS

Table 6.14 reports the total number of correct (TP) and incorrect (TN) predictions for Java-related vulnerabilities.

	# TP	# TN	# total	% TP	% TN	% total
ChatGPT-3.5	165	23	188	87.8%	12.2%	100%
Google Bard	165	23	188	87.8%	12.2%	100%
Android Studio Bot	86	102	188	45.7%	54.3%	100%

Table 6.14: Number of True Positive (TP) and True Negative (TN) comparing LLMs Java-related outputs to ground truth.

6.5.2 XML-RELATED OUTPUTS

Table 6.15 reports the total number of correct (TP) and incorrect (TN) predictions for XML-related vulnerabilities.

	# TP	# TN	# total	% TP	% TN	% total
ChatGPT-3.5	75	21	96	78.1%	21.9%	100%
Google Bard	89	7	96	92.7%	7.3%	100%
Android Studio Bot	10	86	96	10.4%	89.6%	100%

Table 6.15: Number of True Positive (TP) and True Negative (TN) comparing LLMs XML-related outputs to ground truth.

7

Discussion

This Chapter describes all the results shown in the previous one.

7.1 DISCUSSION OF JAVA-RELATED REPAIRS

Starting from Java-related patches, they have been evaluated with CodeBLEU metric. As briefly described in Section 6.1, CodeBLEU is a variation of BLEU metric which is based on n-grams; in particular, CodeBLEU is computed by taking into account four different scores: n-gram match, weighted n-gram match, syntactic match and dataflow match. To give insights about how the final score is computed, in Table 6.2, Table 6.4 and Table 6.6 are reported also the single scores. CodeBLEU score is a value between 0 and 1, where 0 means that the two codes given as input are completely different; on the other hand, the maximum score means that the inputs are identical. The two codes given as input to CodeBLEU are the reference (manually generated ground truth) and the prediction (LLM output) and the resulting score measures the syntactic and semantic similarity between the reference and the prediction. The higher is the similarity score, the higher is the probability that the patch suggested by the large language model is plausible, since it is closer to the ground truth.

To understand how the weights influence the final score, three different contributions are evaluated and reported in Table 7.1 (see Equation 6.1). Starting from codeBLEU_1, all four components have the same weight and equally contribute to the final score. The focus of the evaluation sees the semantic similarity at its core so in codeBLEU_2 and codeBLEU_3, both n-gram and weighted n-gram weights are reduced to give more importance to the syntactic and dataflow match. The three configurations have been evaluated on the all the three LLMs under analysis (ChatGPT, Bard and Studio Bot).

	α	β	γ	δ
codeBLEU_1	0.25	0.25	0.25	0.25
codeBLEU_2	0.15	0.15	0.35	0.35
codeBLEU_3	0.15	0.15	0.3	0.4

Table 7.1: Different weights for computing CodeBLEU.

To grasp more about which can be considered the best trade-off and, more importantly, to define which is the model that performed better, the average and the standard deviation are computed for codeBLEU_1, codeBLEU_2 and codeBLEU_3, for all the three LLMs and showed in Table 7.2 and Table 7.3.

The comparison is always done between LLMs outputs with respect to the same ground truth and the pre-processed outputs are also the same, regardless of the used metric. Starting from Table 7.2, results show that all the three configurations of CodeBLEU have, on average, a higher score for ChatGPT, suggesting that it is the best one in proposing repairs for analysed vulnerabilities and not only in terms of generating patches close to the references. ChatGPT presents the lowest standard deviation values for all the three configurations; thus, its performance results in being stable considering all the different types of vulnerabilities given as input. The distance between ChatGPT, Bard and Studio Bot is consistent among the metrics, having ChatGPT at first place, Bard at second and Studio Bot as the last one. However, there is not so much difference among the evaluated LLMs, in particular between Bard and Studio Bot.

To define which one is the best option is not enough to define the average and the standard deviation, since the dataset is small (188 Java examples) and the number of examples per rule is varying. Enlarging the dataset through the inclusion of more vulnerabilities for rules which have now very few associated examples, may change the final result considering the small difference between the average scores.

Focusing specifically on the evaluated dataset, the higher CodeBLEU score is reached by ChatGPT in all the three different configurations with an average score of about 0.614 obtained with codeBLEU_2 with weights set accordingly as stated in Table 7.1. In general, the average scores for codeBLEU_3, where the highest weight is assigned to the dataflow_match, are not the the best ones, suggesting that balanced weights between dataflow and syntactic match scores may be a valid trade-off. Still, the distance between them is considered minimal and a deeper analysis has to be carried out.

	AVG_codeBLEU_1	AVG_codeBLEU_2	AVG_codeBLEU_3
ChatGPT	0.5839074774	**0.6179233126**	0.6095366419
Bard	0.5576255635	0.5846699652	0.5763806717
Studio Bot	0.5487934192	0.5747148596	0.5683750365

Table 7.2: Averages of the three CodeBLEU configurations for each LLM.

	STDEV_codeBLEU_1	STDEV_codeBLEU_2	STDEV_codeBLEU_3
ChatGPT	0.5839074774	**0.6179233126**	0.6095366419
Bard	0.5576255635	0.5846699652	0.5763806717
Studio Bot	0.5487934192	0.5747148596	0.5683750365

Table 7.3: Standard deviation of the three CodeBLEU configurations for each LLM.

To have a confirmation that actually patches produced by ChatGPT are the ones more similar to ground truth, a human validation has been done to get the number of True Positive (TP) and True Negative (TN) for each tested LLM. Although, ChatGPT and Bard share the same number of correct and incorrect proposed patches (see Table 6.14), their difference in scores is likely due to the similarity of the suggested repairs compared to the ground truth; therefore, even if Bard has a lower CodeBLEU score, it is still a valid solution to adopt for APR tasks.

7.2 DISCUSSION OF XML-RELATED REPAIRS

As done in previous section for Java-related repairs, this section highlights the most relevant results obtained by the analysis carried on patches proposed by LLMs when tested for repairing vulnerabilities extracted from the Android Manifest file.

First of all, a summary of the averages scores and standard deviation from Jaccard, Cosine and Levenshtein metrics are reported in Table 7.4, considering the *pre-processing type 1* described in Section 5.4.2. As a result, from the average scores of each metric Android Studio Bot has the highest similarity score and a very low standard deviation considering cosine similarity measure. All the three metrics evaluations are consistently declaring Studio Bot as the one proposing patches closer to the respective ground truth and the low standard deviation for each metric also confirms such a consistency. However, these results are not reflecting the expectations if the number of True Positive and True Negative are taken into account. Recalling the summary done in Table 6.15, Studio Bot is the worst in generating repairs for XML vulnerabilities, with 89% of incorrect patches. The reason why the evaluation of the three metrics see Studio Bot has the LLM with the best performance is due to the faulty pre-processing; tokenizing the snippets in single words turned out to be a bad approach. Most of the outputs produced by ChatGPT and Bard contain additional information which is useful both for fixing the vulnerability but also for giving more context, which is not interesting for the final objective. For this reason, the number of words composing the outputs by ChatGPT and Bard is definitely higher than the one of Studio Bot, ending up in having fixes proposed by the latter considered more similar to ground truth but only in terms of lengths.

	AVG_jaccard_1	AVG_levenshtein_1	AVG_cosine_1
ChatGPT	0.6520373258	0.5240478429	0.8422516469
Bard	0.6144601183	0.4421176694	0.8037802712
Studio Bot	0.6771177744	0.5939802569	**0.8938985725**

Table 7.4: Average for Jaccard, Cosine and Levenshtein for XML snippets pre-processed with word-tokenizer (*pre-process type 1*).

	STDEV_jaccard_1	STDEV_levenshtein_1	STDEV_cosine_1
ChatGPT	0.2406173385	0.2288010077	0.2238715795
Bard	0.2572517797	0.1982541917	0.3013320102
Studio Bot	0.1902175079	0.1666187735	**0.08027792161**

Table 7.5: Standard Deviation for Jaccard, Cosine and Levenshtein for XML snippets pre-processed with word-tokenizer (*pre-process type 1*).

To support this intuition (confirmed by human validation), another type of pre- processing has been defined as better explained in Section 5.4.2. This focuses only on relevant part of the snippets to be compared, removing useless information added by the LLMs, abstracting arbitrary values and giving the snippets a common format. Table 7.6 and Table 7.7 respectively show the updated averages and standard deviations for the Jaccard, Cosine and Levenshtein metrics after the pre-processing (*type 2*). The average scores and corresponding standard deviations are now reflecting the expectations, with ChatGPT and Bard performing the same on average for all the three metrics, while Studio Bot has the lowest scores. In particular, for Jaccard similarity and Levenshtein distance, ChatGPT reaches on average the highest score; on the other hand, Bard "wins" considering Cosine similarity. However, the difference is definitively small and results can be considered consistent among the metrics and to confirm so, it is possible to notice that the standard deviation for ChatGPT is a little bit higher than the Bard ones, suggesting that even if ChatGPT ends up in having a higher similarity score, this affirmation may be not true if evaluated on more examples; on the other hand, Bard keeps the lowest standard deviation.

	AVG_jaccard_2	AVG_levenshtein_2	AVG_cosine_2
ChatGPT	**0.742248233**	**0.7184974773**	0.8983001031
Bard	0.7372446178	0.6687946861	**0.901329177**
Studio Bot	0.5800937064	0.6196332359	0.611488562

Table 7.6: Average for Jaccard, Cosine and Levenshtein for XML snippets with *pre-process type 2*.

	STDEV_jaccard_2	STDEV_levenshtein_2	STDEV_cosine_2
ChatGPT	0.1901469383	0.2070744872	0.1514960852
Bard	0.1754537826	0.2011573124	**0.1188467039**
Studio Bot	0.2718670314	0.2367921633	0.3475562218

Table 7.7: Standard Deviation for Jaccard, Cosine and Levenshtein for XML snippets with *pre-process type 2*.

To confirm such a discussion, human validation has been performed. Considering Table 6.15, Bard reaches a greater number of True Positive with respect to ChatGPT, suggesting that even if for the latter the average scores are a higher, its higher standard deviation let the results be not so reliable.

7.3 GENERAL DISCUSSION AFTER HUMAN VALIDATION

To have a wider set of results to understand the overall performances of the LLMs, the general Table 6.13 is a summary of the total number of correct patches proposed by the three different LLMs where both Java- and XML-related outputs are taken into account.

Bard results in being the best model for generating repairs because of its better performances for Android Manifest vulnerabilities. Overall, Bard and ChatGPT demonstrated to be able to correctly fix most of the vulnerabilities given as input in a zero-shot scenario, since no fine-tuning and no few-shot learning were applied.

Unexpectedly, Android Studio Bot is not reliable and demonstrated its inability in fixing code. Most of the time, its outputs were a repetition of the code given as input or a natural text affirming that the Bot was not able to understand the code; in other cases, the solution provided was not inherent to the best practice associated with the vulnerable code and which was supposed to guide the LLMs towards the correct implementation of the fix. For instance, Rule 20 is associated with the following best practice: ”*If a service is explicitly exported, declare a permission to protect access to it. If a service declares a permission but it is not explicitly exported, set exported attribute to true.*” and the repair suggested by Studio Bot for most cases was to just set the `android:exported` attribute to ”false”. Although this solution may seem correct, since the access to the component is limited to its own application, it is not what the best practice intended: the component still have to be visible to other applications but accessed only by the ones declaring a specific permission.

Google described Android Studio Bot as a ”*LLM-powered chatbot tightly integrated with Android Studio*” [40] and, for this reason, it was supposed to have the best performance with respect to the other two LLMs, which are general-purposes and not even fine-tuned on a specific set of Android vulnerability examples; on the contrary, it resulted to be the worst one.

7.4 How much reliable are the employed Metrics?

To confirm that considered metrics (CodeBLEU, Jaccard similarity and so on) are actually reliable and related to the results obtain by human validation, a deeper analysis has been conducted. Metrics such as *precision* and *recall* can be more informative regarding the predictions. For instance, precision gives a measure on how many predicted correct patches are actually correct; recall, how many of the True Positive are correctly predicted. These metrics help to understand how well a classifier performs from several point of views.

Focusing on the particular case, metrics used in this project can be treated as ”classifiers” whose role is to give a measure of the similarity between the prediction (output) of the LLM and the ground truth, and state if the output is correct or not. However, the value assigned by the metrics is not binary (e.g 0 for incorrect, 1 for correct), but it is between 0 and 1, which does not allow to have a precise number of False Positive (i.e. predictions considered correct, but are actually incorrect) and False Negative (i.e. predictions considered incorrect, but are actually correct). One strategy may be to set a threshold T and state that for every score greater than T, the prediction is considered correct and incorrect otherwise. Consequently, a possible path to follow is to evaluate the Area Under the Curve (AUC) for the ROC curve (Receiver Operating Characteristic) and for the Precision-Recall curve. In particular, the ROC curve is defined by the True Positive Rate (TPR) and the False Positive Rate (FPR) at different threshold values. The higher is the value of the Area Under the Curve (AUC), the better the classifier is, which means that the value of the area is closer to 1 (perfect classifier, TPR=1 and FPR=0) and not 0.5 (random classifier). The AUC for Precision-Recall curve is, instead, mainly used in a scenario where the performances of the classifier with respect to correct predictions is more relevant. Using a Python library called *sklearn*, the values for ROC-AUC and PR-AUC can be easily computed as shown below:

```python
from sklearn import metrics

roc_auc_score = metrics.roc_auc_score(<y_true>, <y_scores>)
pr_auc_score = metrics.average_precision_score(<y_true>, <y_scores>)
```

Listing 7.1: ROC-AUC and PR-AUC computation.

where *y_true* represents the list of True Positive (i.e. ”1”) and True Negatives (i.e. ”0”) obtained after human validation and *y_scores* are the outputs, in this case, of the metrics.

For instance, to get how well *codeBLEU_1* performed on average, the following steps are executed:

1. compute ROC-AUC and PR-AUC for each pair:
    - (*human_validation*, *codeBLEU_1_chatgpt*)
    - (*human_validation*, *codeBLEU_1_bard*)
    - (*human_validation*, *codeBLEU_1_studiobot*)

    where *codeBLEU_1_<LLM>* is the list of all the similarity scores computed by CodeBLEU (with configuration 1) for the *<LLM>* outputs with respect to the ground truth.
2. once all the pairs have their ROC-AUC and PR-AUC scores, it is possible to compute their average;
3. repeat the process at step 1 for each other metric (*codeBLEU_2*, *codeBLEU_3*, *jaccard_2*, *levenshtein_2*, *cosine_2*).

As a note, metrics used for XML-related outputs, only the ones computed after *pre-process type 2* (i.e. *jaccard_2*, *levenshtein_2*, *cosine_2*) are evaluated in this analysis, since they already resulted to be more reliable and accurate then the ones computed for LLM outputs pre-processed with *pre-process type 1*. Table 7.8 and Table 7.9 show results about the ROC-AUC and PR-AUC scores computed for each configuration of CodeBLEU and for each LLM; with the same reasoning, Table 7.10 and Table 7.11 report results about ROC-AUC and PR-AUC for metrics used for XML-related outputs.

After conducting such an analysis, it is possible to affirm that:

- *codeBLEU_2* has the highest ROC-AUC and PR-AUC values on average and, accordingly to what has been explained before in this section, this result suggests that the metric has a lower rate of False Positive (from ROC-AUC score) and that it has better performances in predicting the positive class (i.e. having correct patches, more similar to the ground truth);
- same reasoning applies to *cosine_2*. It has, on average, the best value for both for ROC-AUC and PR-AUC.

What is stated before can be considered as a confirmation of all the discussions carried out in this Chapter, since from the previous analysis in Section 7.1 and Section 7.2, *codeBLEU_2* and *cosine_2* were already considered as the most performing ones.

Note that in all the tables below *_C* stands for ChatGPT-3.5, *_B* stands for Bard, *_S* stands for Studio Bot.

	ROC-AUC_C	ROC-AUC_B	ROC-AUC_S	ROC-AUC_AVG
codeBLEU_1	0.6005	0.6585	0.6809	0.6466333333
codeBLEU_2	0.5989	0.6519	0.7041	**0.6516333333**
codeBLEU_3	0.5768	0.653	0.7046	0.6448

Table 7.8: ROC-AUC values for CodeBLEU metric (all the configurations).

	PR-AUC_C	PR-AUC_B	PR-AUC_S	PR-AUC_AVG
codeBLEU_1	0.9251	0.931	0.6715	0.8425333333
codeBLEU_2	0.9211	0.931	0.6952	**0.8491**
codeBLEU_3	0.9173	0.931	0.6948	0.8477

Table 7.9: PR-AUC values for CodeBLEU metric (all the configurations).

	ROC-AUC_C	ROC-AUC_B	ROC-AUC_S	ROC-AUC_AVG
jaccard_2	0.8133	0.5819	0.6442	0.6798
levenshtein_2	0.7571	0.4848	0.4587	0.5668666667
cosine_2	0.7971	0.4695	0.8523	**0.7063**

Table 7.10: ROC-AUC values for Jaccard, Levenshtein and Cosine (computed on outputs with *pre-process type 2*).

	PR-AUC_C	PR-AUC_B	PR-AUC_S	PR-AUC_AVG
jaccard_2	0.9342	0.9258	0.1582	0.6727333333
levenshtein_2	0.9041	0.9114	0.1003	0.6386
cosine_2	0.9287	0.9093	0.6785	**0.8388333333**

Table 7.11: PR-AUC values for Jaccard, Levenshtein and Cosine (computed on outputs with *pre-process type 2*).

8

Future Works

This project can be considered as a starting point not only for a more accurate and precise evaluation of LLMs, but also for a larger dataset on which to test them or to build up (train, validate, test) new valuable tools.

Among several future directions and paths to explore, the most relevant ones are cited below:

1. **enlarging the dataset**, by adding new vulnerabilities taken from real-world application as much as it is possible to have more examples for each vulnerability type already included in it. Furthermore, it may be enlarged by synthetically creating new vulnerabilities which violate the best practices not considered in this project, so to cover the highest number best practices - *violated and patched* - defined by Google for Android-based software implementation. Naturally, taking into account the idea of a large dataset collecting general Android vulnerabilities, examples from other domains can be integrated as well;

2. further **exploring the full potentiality of LLMs** and how better they can perform in different scenarios such as *in-context learning*;

3. considering that the behaviour of LLMs is non-deterministic, the overall results may vary; therefore, an additional analysis may include prompting the single LLM with the same prompt multiple times, guaranteeing that no training is active (e.g. in ChatGPT case) and a new chat is started every time. This may lead to a better idea of how the LLM can fix the vulnerable code and if it is able to. For instance,

in the case of an incorrect proposed patch, this may be due to the LLM incapacity or it may just be an isolated case.

4. analyse the performances of the LLMs on single categories of vulnerabilities and not on their totality (all the ones from Java or XML). The analysis on one vulnerability type at a time can highlights their limits and more critical aspects in repairing them, understanding the common underlying pattern of their mistakes while producing the fix.

In general, research about large language models is to be considered promising. The overall evaluation shows how general-purposes LLMs were able to repair most of the vulnerabilities without any additional context; therefore, applying suggestions described above only leads to positive expectations about their possible improvement in performances.

9

Conclusion

In a scenario in which there is a lack of datasets collecting vulnerabilities examples, there is a even deeper unavailability of datasets regarding vulnerabilities with their related fixes and specific Android vulnerabilities. This project aims to fix this gap or, at least, starting to build a new path to do so. A new dataset is proposed and it is based on Android vulnerabilities; it has been developed not only to be a *general* contribution, but mainly to test Large Language Models (LLMs) for code repair tasks from the field of Automated Program Repair (APR); in particular, Android vulnerabilities.

The dataset created is not only a collection of several vulnerabilities, but it also contains related fixes (considered as a ground truth when it comes to evaluate the models). They have been manually implemented and tested, to ensure that for an already successfully compiling application, no compilation errors are going to be introduced by the fix. Once the ground truth for each vulnerability in the dataset has been implemented, the three chosen LLMs (ChatGPT-3.5, Google Bard and Android Studio Bot) have been evaluated by being prompted with the vulnerable code snippet and the respective best practice that the code violates.

Through several similarity metrics, their outputs have been compared with the previously implemented ground truth. The overall results is positive for ChatGPT and Bard, which were able to repair the majority of the vulnerabilities from the created dataset, respectively, 85% and 89% of patches proposed by the two LLMs were set as corrected, after manual validation; on the other hand, Android Studio Bot turned out to have the lowest performances, not being able to correctly repair more than the half of the total vulnerabilities (only 34%). In particular, it has worse performances when trying to fix vulnerabilities extracted from the Android Manifest file. These results suggest that LLMs are not bullet-proof and have to be deeply explored in their limits and a possible way to let their predictions improve. However, considering that they are already capable to understand the problem and to provide a correct solution most of the time even in a zero-shot scenario, including few-shot scenario or fine-tuning them can only lead to positive expectations, where they can achieve high levels of precision and probably being officially a valid - *or maybe better* - alternative to traditional APR tools.

References

[1] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in c code," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 147–165, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2022.3147265

[2] J. Chi, Y. Qu, T. Liu, Q. Zheng, and H. Yin, "Seqtrans: automatic vulnerability fix via sequence to sequence learning," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 564–585, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2022.3156637

[3] M. Monperrus, "Automatic software repair: A bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018. [Online]. Available: https://doi.org/10.1145/3105906

[4] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016. [Online]. Available: https://doi.org/10.1109/TSE.2016.2521368

[5] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang, "A survey on automated program repair techniques," *arXiv preprint arXiv:2303.18184*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.18184

[6] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.18223

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017. [Online]. Available: https://doi.org/10.5555/3295222.3295349

[8] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019. [Online]. Available: https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[9] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020. [Online]. Available: https://doi.org/10.48550/arXiv.1910.10683

[10] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," 2021, p. 8696 – 8708. [Online]. Available: https://doi.org/10.48550/arXiv.2109.00859

[11] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," *arXiv preprint arXiv:2002.08155*, 2020. [Online]. Available: https://doi.org/10.48550/arXiv.2002.08155

[12] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "GraphCodeBERT: Pre-training Code Representations with Data Flow," *arXiv preprint arXiv:2009.08366*, 2020. [Online]. Available: https://doi.org/10.48550/arXiv.2009.08366

[13] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959–971. [Online]. Available: https://doi.org/10.1145/3540250.3549101

[14] "Google Bard." [Online]. Available: https://blog.google/technology/ai/bard-google-ai-search-updates/

[15] "Android Studio Bot." [Online]. Available: https://developer.android.com/studio/preview/studio-bot

[16] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the aaai conference*

on artificial intelligence, vol. 31, no. 1, 2017. [Online]. Available: https://doi.org/10.1609/aaai.v31i1.10742

[17] “Global Market Held By Mobile Operating Systems.” [Online]. Available: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[18] “App Security Best Practices.” [Online]. Available: https://developer.android.com/privacy-and-security/security-tips

[19] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, “A survey of learning-based automated program repair,” *ACM Trans. Softw. Eng. Methodol.*, 2023. [Online]. Available: https://doi.org/10.1145/3631974

[20] M. Monperrus, “The Living Review on Automated Program Repair,” 2020. [Online]. Available: https://hal.science/hal-01956501

[21] C. S. Xia, Y. Wei, and L. Zhang, “Automated program repair in the era of large pre-trained language models,” in *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00129

[22] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2107.03374

[23] H. Joshi, J. C. Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, “Repair is nearly generation: Multilingual program repair with llms,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 4, 2023, pp. 5131–5140. [Online]. Available: https://doi.org/10.48550/arXiv.2208.11640

[24] N. Jiang, K. Liu, T. Lutellier, and L. Tan, “Impact of code language models on automated program repair,” 2023, p. 1430 – 1442. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00125

[25] C. S. Xia and L. Zhang, “Conversational automated program repair,” *arXiv preprint arXiv:2301.13246*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2301.13246

[26] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 935–947. [Online]. Available: https://doi.org/10.1145/3540250.3549098

[27] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2339–2356. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.10179420

[28] O. Lieber, O. Sharir, B. Lenz, and Y. Shoham, "Jurassic-1: Technical details and evaluation," *White Paper. AI21 Labs*, vol. 1, 2021.

[29] S. Feng and C. Chen, "Prompting is all your need: Automated android bug replay with large language models," *arXiv preprint arXiv:2306.01987*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.01987

[30] Y. Zhao, T. Su, Y. Liu, W. Zheng, X. Wu, R. Kavuluru, W. G. Halfond, and T. Yu, "Recdroid+: Automated end-to-end crash reproduction from bug reports for android apps," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–33, 2022. [Online]. Available: https://doi.org/10.1145/3488244

[31] J. Johnson, J. Mahmud, T. Wendland, K. Moran, J. Rubin, and M. Fazzini, "An empirical investigation into the reproduction of bug reports for android apps," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 321–322. [Online]. Available: https://doi.org/10.1109/SANER53432.2022.00048

[32] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 23–26. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.8

[33] Z. Namrud, S. Kpodjedo, and C. Talhi, "Androvul: A repository for android security vulnerabilities," in *Proceedings of the 29th Annual International*

Conference on Computer Science and Software Engineering, 2019, pp. 64–71. [Online]. Available: htpps://doi.org/10.5555/3370272.3370279

[34] "AndroBugs Framework as an Android Vulnerability Analysis System." [Online]. Available: https://www.androbugs.com/

[35] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th international conference on mining software repositories*, 2016, pp. 468–471.

[36] J. Gajrani, M. Tripathi, V. Laxmi, G. Somani, A. Zemmari, and M. S. Gaur, "Vulvet: Vetting of vulnerabilities in android apps to thwart exploitation," *Digital Threats: Research and Practice*, vol. 1, no. 2, pp. 1–25, 2020. [Online]. Available: https://doi.org/10.1145/3376121

[37] "F-Droid: Free and Open Source Android App Repository." [Online]. Available: https://f-droid.org/

[38] "SPECK: From Google Textual Guidelines to Automatic Detection of Android Apps Vulnerabilities." [Online]. Available: https://github.com/SPRITZ-Research-Group/SPECK

[39] "JADX: Dex to Java Decompiler." [Online]. Available: https://github.com/skylot/jadx

[40] "Android Studio Bot as a LLM-powered chatbot." [Online]. Available: https://developer.android.com/studio/preview/studio-bot#other-chatbots

[41] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati, "Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks," *arXiv preprint arXiv:2310.10508*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.10508

A

Appendix A

Appendix A provides the list of all the applications analysed for the realisation of the dataset.

Applications List		
a2dpvolume	a2linux	aagtl
aard2	addi	adsdroid
agoradesk	aGrep	air_guard
airmessage	alarmclock	aLogCat
alovoa	amazed	androidomatic-keyer
anonymous-messenger	antennapod	anycut
anymemo	apg-master	apk-mirror
appcachecleaner	aruba-networks-login	arxiv-papers-mobile
audio-share	autoanswer	babyphone
barinsta	basedcooking	batterydog
bites	blokish	blood-pressure-monitor
bluepass	bookcatalogue	book-listing-app
camera-align	canta	catima
cfc-master	clima	collins-dictionary
content-blocker	cryptomator	dawdle
deepl	destiny	duckduckgo

Applications List		
easy_xkcd	ebook-downloader	element
evmap	fake-traveler	fdroidclient
flipper	fluffychat	frozenbubble
gpt-assist	hacki	hex-viewer
homeassistant	hotdeath	k9
keepass2android	kpassnotes	leon
librepass	linuxcommandlibrary	localsend
lockpatterngenerator	logmein	lolcatbuilder
lucia-app	manpages	midori
mileage	mininoteviewer	monocles-chat
mullvadvpn	multismssender	munchlife
myexpenses	mylock	nectroid
notesnook	omweather	opencamera-code
open-hub	open-in-wa	openrecipes
opentrack	passport-reader	pass-store
passwdsafe	passwordmaker	passy
paynoway	pcapdroid	pincredible
privacycell	quickmsg	recordyou
remote-video-cam	sanity	savertuner
seguime	share-my-number	sharex
silence	simple-login	simplex-chat-stable
simplytranslate_mobile	sleepy-wifi	smart-cookies-web
sound-recorder	squawker	status-mobile
stoic_reading	swiftp	telegram-android
telegram-foss	tice	tinfoil-twitter
tiny-travel-tracker	tracker-control	twire
unstoppable-wallet	vigilante	water-me
watomatic	webcall	webmon
weight-chart	whohasmystuff	wikipedia
windscribe	word-of-the-day	worldclock

B

Appendix B

The Appendix B wants to illustrate examples of vulnerable code and respective ground truth for each best practice such that a violation is included in the dataset. For each ”Rule” (best practice), two examples are shown; in case of rules for which only one violation has been found, it is reported as well.

B.1 RULE 1: SHOW AN APP CHOOSER

Best practice: ”Show an app chooser. Implicit intents may be handled by more than one activity, therefore show an app chooser before launching the intent.”

```
    public void onCreate(Bundle icicle) {
        // rest of the code
        final Intent intent = getIntent();
        if (intent.getData() == null)  {
            intent.setData(Provider.Recipes.CONTENT_URI);
        }
        action = intent.getAction();
        // rest of the code
    }
```

Listing B.1: Vulnerable code for 1.

```java
// fix1: show an app chooser for implicit intent which may be handled by more than one activity
// check if the number of possible activities that can manage the intent is greater than 1, if so, show an app chooser

public void onCreate(Bundle icicle) {
    // rest of the code
    final Intent intent = getIntent();
    if (intent.getData() == null)  {
        intent.setData(Provider.Recipes.CONTENT_URI);
    }
    action = intent.getAction();
    // fix1
    List<ResolveInfo> possibleActivitiesList = getPackageManager().queryIntentActivities(intent, PackageManager.MATCH_ALL);
    if (possibleActivitiesList.size() > 1) {
        String title = getResources().getString(R.string.chooser_title);
        Intent chooser = Intent.createChooser(intent, title);
        startActivity(chooser);
    }
    // rest of the code
}
```

Listing B.2: Ground truth for 1.

```java
public void onClickFilesSelect()  {
    Intent i = new Intent(Intent.ACTION_GET_CONTENT);
    i.setType("*/*");
    i.putExtra(Intent.EXTRA_ALLOW_MULTIPLE, true);
    mutipleFilesActivityResultLauncher.launch(i);
    // rest of the code
}
```

Listing B.3: Vulnerable code for 1.

```java
// fix1: show an app chooser for implicit intent which may be handled by more than one activity
// check if the number of possible activities that can manage the intent is greater than 1, if so, show an app chooser

public void onClickFilesSelect()  {
    Intent i = new Intent(Intent.ACTION_GET_CONTENT);
```

```
        i.setType("*/*");
        i.putExtra(Intent.EXTRA_ALLOW_MULTIPLE, true);
        // fix1
        List<ResolveInfo> possibleActivitiesList = getPackageManager().
    queryIntentActivities(i, PackageManager.MATCH_ALL);
        if (possibleActivitiesList.size() > 1) {
            String title = getResources().getString(R.string.chooser_title);
            Intent chooser = Intent.createChooser(i, title);
            startActivity(chooser);
        } else {
            mutipleFilesActivityResultLauncher.launch(i);
        }
        // rest of the code
    }
```

Listing B.4: Ground truth for 1.

B.2 RULE 3: REMOVE UNUSED PERMISSIONS

Best practice: ”Remove unused permissions. They are declared in the manifest but there are no components that need them.”

```
<uses-permission android:name="android.permission.CHANGE_WIFI_MULTICAST_STATE"
    />
```

Listing B.5: Vulnerable code 1 for Rule 3.

```
<!--fix3: the permission is never called or used in the app, no need to declare
     it-->
<!--delete or comment the permission-->
<!--<uses-permission android:name="android.permission.
    CHANGE_WIFI_MULTICAST_STATE" />-->
```

Listing B.6: Ground truth 1 for Rule 3.

```
<uses-permission android:name="android.permission.WAKE_LOCK"></uses-permission>
```

Listing B.7: Vulnerable code 2 for Rule 3.

```
<!--fix3: the permission is never called or used in the app, no need to declare
     it-->
<!--delete or comment the permission-->
```

```
<!--<uses-permission android:name="android.permission.WAKE_LOCK"></uses-
    permission>-->
```

Listing B.8: Ground truth 2 for Rule 3.

B.3 RULE 4: USE INTENTS TO DEFER PERMISSIONS

Best practice: ”Use intents to defer permissions. There are several dangerous permissions that may be avoided just by using intents to perform the same operation.”

```
public void sendMessage(Handler handler) {
    // rest of the code
    for (int i = message_sent; i < chunk_max; i++) {
        message_sent++;
        String newN = numbers[i];
        Message msg = handler.obtainMessage();
        Bundle b = new Bundle();
        b.putInt("ORIGIN", DIALOG_PROGRESS);
        b.putInt("total", (i * 100) / size);
        msg.setData(b);
        handler.sendMessage(msg);
        ArrayList<PendingIntent> deliveryIntents = null;
        ArrayList<PendingIntent> sentIntents = null;
        if (haveDeliveryReports) {
            deliveryIntents = new ArrayList<PendingIntent>(messageCount);
            sentIntents = new ArrayList<PendingIntent>(messageCount);
            ContentValues values = new ContentValues();
            values.put("address", newN);
            values.put("body", message);
            getContentResolver().insert(Uri.parse("content://sms/sent"), values
    );
            long entryId = deliveryIdMap.get(newN);
            for (int j = 0; j < messageCount; j++) {
                if (j == (messageCount - 1)) {
                    Uri entryURI = Uri.withAppendedPath(DeliveryDbAdapter.
    CONTENT_DELIVERY_URI, "" + entryId);
                    Intent intent = new Intent(MessageReceiver.MESSAGE_RECEIVED
    , entryURI, this, MessageReceiver.class);
                    deliveryIntents.add(PendingIntent.getBroadcast(this, 0,
    intent, 0));
```

```java
                } else {
                    deliveryIntents.add(null);
                }
            }
        }
        manager.sendMultipartTextMessage(newN, null, messages, sentIntents,
    deliveryIntents);
    }
    // rest of the code
}
```

Listing B.9: Vulnerable code for Rule 4.

```java
// fix4: use intents instead of dangerous permission

public void sendMessage(Handler handler) {
    // rest of the code
    for (int i = message_sent; i < chunk_max; i++) {
        message_sent++;
        String newN = numbers[i];
        Message msg = handler.obtainMessage();
        Bundle b = new Bundle();
        b.putInt("ORIGIN", DIALOG_PROGRESS);
        b.putInt("total", (i * 100) / size);
        msg.setData(b);
        handler.sendMessage(msg);
        // rest of the code
        // fix4
        Intent sendMsg = new Intent(this, MessageReceiver.class);
        sendMsg.setAction(Intent.ACTION_SENDTO);
        sendMsg.setData(Uri.parse("smsto:" + newN));
        sendMsg.putExtra("sms_body", messages);
        startActivity(sendMsg);

        manager.sendMultipartTextMessage(newN, null, messages, sentIntents,
    deliveryIntents);
    }
    // rest of the code
}
```

Listing B.10: Ground truth for Rule 4.

B.4 RULE 5: USE SSL TRAFFIC

Best practice: ”Do not use HttpURLConnection but its secure version HttpsURLConnection and handle possible risk of SSLHandshakeException.”

```
private InputStream doPost2(String urlString, List<NameValuePair> values,
    ByteArrayOutputStream content) throws IOException   {
    URL url = new URL(urlString);
    HttpURLConnection con = (HttpURLConnection) url.openConnection();
    // rest of the code
}
```

Listing B.11: Vulnerable code 1 for Rule 5.

```
// fix5: do not use HttpURLConnection, but its secure version
    HttpsURLConnection
// NOTE: add the necessary imports and handle SSLHandshakeException in a try-
    catch block

private InputStream doPost2(String urlString, List<NameValuePair> values,
    ByteArrayOutputStream content) throws IOException   {
    URL url = new URL(urlString);
    // fix5
    try {
        HttpsURLConnection con = (HttpsURLConnection) url.openConnection();
    } catch (SSLHandshakeException e)  {
        e.printStackTrace();
    }
    // rest of the code
}
```

Listing B.12: Ground truth 1 for Rule 5.

```
private StatusCode login_runner(String username, String password) throws
    Exception  {
    // rest of the code
    URL puServerUrl = new URL(request);
    URLConnection puServerConnection = puServerUrl.openConnection();
    // rest of the code

}
```

Listing B.13: Vulnerable code 2 for Rule 5.

```
// fix5: do not use HttpURLConnection, but its secure version
    HttpsURLConnection
// NOTE: add the necessary imports and handle SSLHandshakeException in a try-
    catch block

private StatusCode login_runner(String username, String password) throws
    Exception  {
    // rest of the code
    URL puServerUrl = new URL(request);
    // fix5: declare "puServerConnection" as HttpsURLConnection and handle
    SSLHandshakeException
    try {
        HttpsURLConnection puServerConnection = (HttpsURLConnection)
    puServerUrl.openConnection();
    } catch (SSLHandshakeException e) {
        e.printStackTrace();
    }
    // rest of the code

}
```

Listing B.14: Ground truth 2 for Rule 5.

B.5 RULE 6: USE JAVASCRIPT CAREFULLY

Best practice: ”Do not enable Javascript interface for a WebView if Javascript is not allowed. Enable it only if the Javascript interface is safe to use.”

```
public ArticleWebView(Context context, AttributeSet attrs)  {
    super(context, attrs);
    // rest of the code
    WebSettings settings = this.getSettings();
    settings.setJavaScriptEnabled(true);
    // rest of the code
    this.addJavascriptInterface(this, "$SLOB");
    // rest of the code
}
```

Listing B.15: Vulnerable code 1 for Rule 6.

```java
// fix6: pay attention when using Javascript interface along with a WebView
// enable it only if necessary

public class ArticleWebView extends WebView {

    public ArticleWebView(Context context, AttributeSet attrs)  {
        super(context, attrs);
        // rest of the code
        WebSettings settings = this.getSettings();
        // fix6: add an if block to check if the condition is satisfied and JS
    is safe to use
        if (isJSAllowed()) {
            settings.setJavaScriptEnabled(true);
            // rest of the code
            this.addJavascriptInterface(this, "$SLOB");
        }

        // rest of the code
    }

    // fix6: define here your condition
    private boolean isJSAllowed() {
        // condition
        return true;
    }

    // rest of the code
}
```

Listing B.16: Ground truth 1 for Rule 6.

```java
private void init(AttributeSet attrs)  {
    // rest of the code
    WebSettings settings = getSettings();
    settings.setJavaScriptEnabled(true);
    // rest of the code
}
```

Listing B.17: Vulnerable code 2 for Rule 6.

```java
// fix6: pay attention when using Javascript interface along with a WebView
// enable it only if necessary
```

```

private void init(AttributeSet attrs)  {
    // rest of the code
    WebSettings settings = getSettings();
    // fix6: add an if block to check if the condition is satisfied and JS is
    safe to use
    if (isJSAllowed()) {
        settings.setJavaScriptEnabled(true);
    }
    // rest of the code
}

// fix6: define here your condition
private boolean isJSAllowed() {
    // condition
    return true;
}
```

Listing B.18: Ground truth 2 for Rule 6.

B.6 RULE 7: WHITELIST URL LOADED IN WEBVIEW

Best practice: ”Do not load urls into a WebView if they are not from a whitelist.”

```
// rest of the code
webView = findViewById(R.id.webView0);
webView.loadUrl("file:///android_asset/map.html");
// rest of the code
```

Listing B.19: Vulnerable code 1 for Rule 7.

```
// fix7: add the .setWebViewClient() to override the shouldOverrideUrlLoading
    method and
// check whether the URL is safe to load (it belongs to a whitelist)

webView = findViewById(R.id.webView0);
// rest of the code
// fix7: add .setWebViewClient
webView.setWebViewClient(new MyWebViewClient() {
    @Override
```

```
    public boolean shouldOverrideUrlLoading(WebView view, WebResourceRequest
    request) {
        String url = request.getUrl().toString();
        // fix7: check if the URL is in a whitelist
        if (isAllowed(url)) {
            //the url to be loaded with webview is ok
            return false;
        } else { return true; }     //blocking the url - overriding
    }

    // fix7: define here your whitelist
    private boolean isAllowed(String url) {
        Set<String> allowed_urls = new HashSet<>();
        // add elements to the set
        if (allowed_urls.contains(url)) {
            return true;
        } else { return false; }

    }
});
// rest of the code
webView.loadUrl("file:///android_asset/map.html");
```

Listing B.20: Ground truth 1 for Rule 7.

```
// rest of the code
myWebView.setVisibility(View.VISIBLE);
myNewWebView.setVisibility(View.INVISIBLE);
myNewWebView.loadUrl("about:blank");
return;
// rest of the code
```

Listing B.21: Vulnerable code 2 for Rule .

```
// fix7: add the .setWebViewClient() to override the shouldOverrideUrlLoading
    method and
// check whether the URL is safe to load (it belongs to a whitelist)

// rest of the code
myNewWebView.setVisibility(View.INVISIBLE);
// fix7: add .setWebViewClient
myNewWebView.setWebViewClient(new MyWebViewClient() {
```

```
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, WebResourceRequest
    request) {
        String url = request.getUrl().toString();
        // fix7: check if the URL is in a whitelist
        if (isAllowed(url)) {
            //the url to be loaded with webview is ok
            return false;
        } else { return true; }     //blocking the url - overriding
    }

    // fix7: define here your whitelist
    private boolean isAllowed(String url) {
        Set<String> allowed_urls = new HashSet<>();
        // add elements to the set
        if (allowed_urls.contains(url)) {
            return true;
        } else { return false; }
    });
    // rest of the code
}
myNewWebView.loadUrl("about:blank");
return;
// rest of the code
```

Listing B.22: Ground truth 2 for Rule .

B.7 RULE 9: SHARE DATA SECURELY ACROSS APPLICATIONS

Best practice: ”When using intents, share data carefully.”

```
public void onFileSelected(File file, boolean checked)  {
    Intent intent = new Intent();
    intent.setData(Uri.fromFile(file));
    fragment.onActivityResult(requestCode, Activity.RESULT_OK, intent);
}
// rest of the code
```

Listing B.23: Vulnerable code 1 for Rule 9.

```
// fix9: guarantee one-time access to data by using
    FLAG_GRANT_READ_URI_PERMISSION and FLAG_GRANT_WRITE_URI_PERMISSION flags
// and do not use file:// to access files, but content://

public void onFileSelected(File file, boolean checked)  {
    Intent intent = new Intent();
    // fix9
    Uri contentUri = FileProvider.getUriForFile(this, "file_provider", file);
    Uri parse = Uri.parse(contentUri);
    intent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
    intent.addFlags(Intent.FLAG_GRANT_WRITE_URI_PERMISSION);
    intent.setData(parse);
    fragment.onActivityResult(requestCode, Activity.RESULT_OK, intent);
}
// rest of the code
```

Listing B.24: Ground truth 1 for Rule 9.

B.8 RULE 15: AVOID CUSTOM DANGEROUS PERMISSION

Best practice: ”Do not declare custom permissions with a dangerous protection level.”

```
    // rest of the code
    android:name="${applicationId}.READ_CARDS"
    android:protectionLevel="dangerous" />
```

Listing B.25: Vulnerable code 1 for Rule 15.

```
<!--fix15: do not declare custom permissions with protectionLevel "dangerous",
    choose "normal" or "signature" instead-->
<permission
    android:description="@string/permissionReadCardsDescription"
    android:protectionLevel="signature" <!--fix15--> />
```

Listing B.26: Ground truth 1 for Rule 15.

B.9 RULE 16: ERASE DATA IN WEBVIEW CACHE

Best practice: "Always clean the WebView cache."

```java
public void onCreate(Bundle savedInstanceState) {
    // rest of the code
    WebView webView = (WebView)findViewById(R.id.about_webview);
    // rest of the code
}
```

Listing B.27: Vulnerable code 1 for Rule 16.

```java
// fix16: clear your webview cache
public void onCreate(Bundle savedInstanceState)     {
    // rest of the code
    WebView webView = (WebView)findViewById(R.id.about_webview);
    // rest of the code
    // fix16
    webView.clearCache(true);
}
```

Listing B.28: Ground truth 1 for Rule 16.

```java
// rest of the code
myNewWebView.setVisibility(View.VISIBLE);
myNewWebView.setVisibility(View.INVISIBLE);
myNewWebView.loadUrl("about:blank");
return;
// rest of the code
```

Listing B.29: Vulnerable code 2 for Rule 16.

```java
// fix16: clear your webview cache
// rest of the code
myWebView.setVisibility(View.VISIBLE);
myNewWebView.setVisibility(View.INVISIBLE);
myNewWebView.loadUrl("about:blank");
// fix16
myNewWebView.clearCache(true);
return;
// rest of the code
```

Listing B.30: Ground truth 2 for Rule 16.

B.10 Rule 17: Avoid SQL injections

Best practice: ”If query(), update(), delete() methods from ContentProvider interface are overridden, check if selection is concatenated with user data and may cause SQL injection.”

```java
public Cursor query(Uri uri, String[] projection, String selection, String[]
    selectionArgs, String sortOrder) {
    // rest of the code
    case ALARM_ID:
        alarmid = ContentUris.parseId(uri);
        c = db.query(AlarmEntry.TABLE_NAME, projection, AlarmEntry._ID + " == "
     + alarmid, null, null, null, null);
        c.setNotificationUri(getContext().getContentResolver(), uri);
        return c;
    // rest of the code
}
```

Listing B.31: Vulnerable code 1 for Rule 17.

```java
// fix17: when overriding methods from ContentProvider class, pay attention not
     to concatenate selection with user data

public Cursor query(Uri uri, String[] projection, String selection, String[]
    selectionArgs, String sortOrder)  {
    // rest of the code
    case ALARM_ID:
        alarmid = ContentUris.parseId(uri);
        // fix17: use placeholders
        c = db.query(AlarmEntry.TABLE_NAME, projection, AlarmEntry._ID + " = ?
    ", new String[] {String.valueOf(alarmid)}, null, null, null);
        c.setNotificationUri(getContext().getContentResolver(), uri);
        return c;
        // rest of the code
}
```

Listing B.32: Ground truth 1 for Rule 17.

```java
public int delete(Uri uri, String selection, String[] selectionArgs)  {
    // rest of the code
    String id = uri.getPathSegments().get(1);
```

```java
    count = db.delete(DATABASE_DELIVERY_ENTRY_TABLE, KEY_DELIVERY_ENTRY_ROWID + "=" + id + (!TextUtils.isEmpty(selection) ? " AND (" + selection + ')' : ""), selectionArgs);
    // rest of the code
}
```

Listing B.33: Vulnerable code 2 for Rule 17.

```java
// fix17: when overriding methods from ContentProvider class, pay attention not to concatenate selection with user data

public int delete(Uri uri, String selection, String[] selectionArgs)  {
    // rest of the code
    String id = uri.getPathSegments().get(1);
    // fix17: use placeholders and merge with "selectionArgs" argument
    try {
        List<String> selArgs = new ArrayList<>(Arrays.stream(selectionArgs).toList());
        String args = !TextUtils.isEmpty(selection) ? " AND (" + selection + ')' : "");
        selArgs.add(String.valueOf(id));
        selArgs.add(args);
        String[] new_selectionArgs = (String[]) selArgs.toArray();
        count = db.delete(DATABASE_DELIVERY_ENTRY_TABLE, KEY_DELIVERY_ENTRY_ROWID + " = ?", new_selectionArgs);
    } catch(ClassCastException ignored) {}
        count = db.delete(DATABASE_DELIVERY_ENTRY_TABLE, KEY_DELIVERY_ENTRY_ROWID + " = ?", selectionArgs);
        // rest of the code
}
```

Listing B.34: Ground truth 2 for Rule 17.

B.11 RULE 18: PREFER EXPLICIT INTENTS

Best practice: ”Do not use implicit intents for startService(), bindService(), startActivity() and sendOrderdBroadcast(); use explicit intents.”

```java
public void onCreate()  {
    // rest of the code
    Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse(st));
```

```
    intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
    startActivity(intent);
    // rest of the code
}
```

Listing B.35: Vulnerable code 1 for Rule 18.

```
// fix18: if an intent is launched with startService(), startActivity(),
    bindService(), sendOrderedBroadcast() set it as explicit intent
// Intent intent = new Intent(<this_context>, <target_class.class>)

public void onCreate()  {
    // rest of the code
    // fix18
    Intent intent = new Intent(this, target_class.class);
    intent.setAction(Intent.ACTION_VIEW);
    intent.setData(Uri.parse(st));
    intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
    startActivity(intent);
    // rest of the code
}
```

Listing B.36: Ground truth 1 for Rule 18.

```
// rest of the code
Intent buttonUp = new Intent(Intent.ACTION_MEDIA_BUTTON);
buttonUp.putExtra(Intent.EXTRA_KEY_EVENT, new KeyEvent(KeyEvent.ACTION_UP,
    KeyEvent.KEYCODE_HEADSETHOOK));
context.sendOrderedBroadcast(buttonUp, "android.permission.CALL_PRIVILEGED");
// rest of the code
```

Listing B.37: Vulnerable code 2 for Rule 18.

```
// fix18: if an intent is launched with startService(), startActivity(),
    bindService(), sendOrderedBroadcast() set it as explicit intent
// Intent intent = new Intent(<this_context>, <target_class.class>)

// rest of the code
// fix18
Intent buttonUp = new Intent(this, target_class.class);
buttonUp.setAction(Intent.ACTION_MEDIA_BUTTON);
buttonUp.putExtra(Intent.EXTRA_KEY_EVENT, new KeyEvent(KeyEvent.ACTION_UP,
    KeyEvent.KEYCODE_HEADSETHOOK));
```

```
context.sendOrderedBroadcast(buttonUp, "android.permission.CALL_PRIVILEGED");
// rest of the code
```

Listing B.38: Ground truth 2 for Rule 18.

B.12 RULE 20: PROTECT SERVICES

Best practice: ”If a service is explicitly exported, declare a permission to protect access to it. If a service declares a permission but it is not explicitly exported, set exported attribute to true.”

```
<service
    android:name=".NotificationCatcher"
    android:permission="android.permission.BIND_NOTIFICATION_LISTENER_SERVICE">
</service>
```

Listing B.39: Vulnerable code 1 for Rule 20.

```
<!--fix20: if a service is declaring a permission, explicitely set it as
    exported-->
<service
android:name=".NotificationCatcher"
android:permission="android.permission.BIND_NOTIFICATION_LISTENER_SERVICE"
android:exported = "true" <!--fix20--> >
</service>
```

Listing B.40: Ground truth 1 for Rule 20.

```
<service
    android:name=".onboarding.WearOnboardingListener"
    android:exported="true">
</service>
```

Listing B.41: Vulnerable code 2 for Rule 20.

```
<!--fix20: if a service is declaring a permission, explicitely set it as
    exported-->
<!--define your permission here-->
<permission
    android:name="custom_permission"
    android:protectionLevel="normal"/>
<service
```

```
    android:name=".onboarding.WearOnboardingListener"
    android:exported="true"
    android:permission="custom_permission" <!--choose your permission here--> >
</service>
```

Listing B.42: Ground truth 2 for Rule 20.

B.13 RULE 22: USE *SECURERANDOMGENERATOR*

Best practice: ”Use a Secure Random Generator to generate keys.”

```java
private static byte[] generateEncryptionSecret()  {
    try  {
        KeyGenerator generator = KeyGenerator.getInstance("AES");
        generator.init(128);
        SecretKey key = generator.generateKey();
        return key.getEncoded();
    } catch (NoSuchAlgorithmException ex)  {
        Log.w("keyutil", ex);
        return null;
    }
}
```

Listing B.43: Vulnerable code 1 for Rule 22.

```java
// fix22: use SecureRandom to initialise cryptographic keys generated by
    KeyGenerator

private static byte[] generateEncryptionSecret()  {
    try  {
        // fix22
        SecureRandom secure_rnd = SecureRandom.getInstanceStrong();
        // choose encryption algorithm
        KeyGenerator key_gen = KeyGenerator.getInstance("AES");
        // choose key size
        key_gen.init(128, secure_rnd);
        Key key = key_gen.generateKey();
    } catch (NoSuchAlgorithmException ex)  {
        Log.w("keyutil", ex);
        return null;
    }
```

```java
}
```

Listing B.44: Ground truth 1 for Rule 22.

```java
public static int calcMaxKeySize()   {
    int[] keySizes = new int[]{256, 192, 128};
    for (int keySize : keySizes) {
        try  {
            KeyGenerator keyGenerator = KeyGenerator.getInstance(INTERNAL_SYMMETRIC_ENCRYPTION_NAME);
            keyGenerator.init(keySize);
            SecretKey key = keyGenerator.generateKey();
            Cipher cipher = Cipher.getInstance(INTERNAL_SYMMETRIC_ENCRYPTION_ALGORITHM);
            cipher.init(Cipher.ENCRYPT_MODE, key);
        }
        catch(Exception e) {
            Log.d(GTG.TAG, "can't use keysize " + keySize + ": " + e);
            continue;
        }
        return keySize;
    }
    // rest of the code
}
```

Listing B.45: Vulnerable code 2 for Rule 22.

```java
// fix22: use SecureRandom to initialise cryptographic keys generated by KeyGenerator

public static int calcMaxKeySize()   {
    int [] keySizes = new int []  {256, 192, 128 };
        for(int keySize : keySizes)     {
            try  {
                // fix22
                SecureRandom secure_rnd = SecureRandom.getInstanceStrong();
                // choose encryption algorithm
                KeyGenerator key_gen = KeyGenerator.getInstance("AES");
                key_gen.init(keySize, secure_rnd);
                Key key = key_gen.generateKey()
                // rest of the code
            }
```

```
            catch(Exception e)       {
                Log.d(GTG.TAG, "can't use keysize "+keySize+": "+e);
                continue;
            }
        }
        return keySize;
    // rest of the code
}
```

Listing B.46: Ground truth 2 for Rule 22.

B.14 RULE 23: PROTECT RECEIVERS

Best practice: ”If a receiver is explicitly exported, declare a permission to protect access to it.”

```
<receiver android:name=".ApplyMockBroadcastReceiver"  android:exported="true">
```

Listing B.47: Vulnerable code 1 for Rule 23.

```
<!--fix23: if a receiver is exported, explicitely add a permission to protect
    it-->
<!--define your permission here-->
<permission
    android:name="custom_permission"
    android:protectionLevel="normal"/>
<receiver
    android:name=".ApplyMockBroadcastReceiver"
    android:exported="true"
    android:permission="custom_permission" <!--choose your permission here-->
    />
```

Listing B.48: Ground truth 1 for Rule 23.

```
<receiver
    android:name="org.totschnig.myexpenses.widget.TemplateWidget"
    android:exported="true"
    android:label="@string/widget_title_templates" >
    <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
    </intent-filter>
```

```
    <meta-data
        android:name="android.appwidget.provider"
        android:resource="@xml/widget_info_generic" />
</receiver>
```

Listing B.49: Vulnerable code 2 for Rule 23.

```
<!--fix23: if a receiver is exported, explicitely add a permission to protect it-->
<!--define your permission here-->
<permission android:name="custom_permission" android:protectionLevel="normal"/>
<receiver
    android:name="org.totschnig.myexpenses.widget.TemplateWidget"
    android:exported="true"
    android:label="@string/widget_title_templates"
    android:permission="custom_permission" <!--choose your permission here--> >
    <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
    </intent-filter>
    <meta-data
        android:name="android.appwidget.provider"
        android:resource="@xml/widget_info_generic" />
</receiver>
```

Listing B.50: Ground truth 2 for Rule 23.

B.15 Rule 26: Set Hostname Verifier for SSLSockets

Best practice: ”Perform Hostname verification if using SSL Sockets, since they don’t do it by default; perform the verification preferably by calling getDefaultHostnameVerifier() with the expected hostname.”

```
public Socket createSocket() throws IOException {
    SSLSocket result = (SSLSocket) factory.createSocket();
    configureSocket(result);
    return result;
}
```

Listing B.51: Vulnerable code 1 for Rule 26.

```
// fix26: perfom HostName verification through HostNameVerifier
```

```java
// NOTE: add necessary imports

public Socket createSocket() throws IOException  {
    SSLSocket result = (SSLSocket) factory.createSocket();

    // fix26
    HostnameVerifier hostname_verif = HttpsURLConnection.
    getDefaultHostnameVerifier();
    boolean is_verified = hostname_verif.verify(result, result.getSession());
    if (is_verified) {
        // write what to do
    }
    else {
        // write what to do
    }

    configureSocket(result);
    return result;
}
```

Listing B.52: Ground truth 1 for Rule 26.

```java
public Socket createSocket(InetAddress var1, int var2, InetAddress var3, int
    var4) throws IOException  {
    SSLSocket result = (SSLSocket) factory.createSocket(var1, var2, var3, var4)
    ;
    configureSocket(result);
    return result;
}
```

Listing B.53: Vulnerable code 2 for Rule 26.

```java
// fix26: perfom HostName verification through HostNameVerifier
// NOTE: add necessary imports

public Socket createSocket(InetAddress var1, int var2, InetAddress var3, int
    var4) throws IOException  {
    SSLSocket result = (SSLSocket) factory.createSocket(var1, var2, var3, var4)
    ;
    // fix26
    HostnameVerifier hostname_verif = HttpsURLConnection.
    getDefaultHostnameVerifier();
```

```
    boolean is_verified = hostname_verif.verify(result, result.getSession());
    if (is_verified) {
        // write what to do
    }
    else {
        // write what to do
    }
    configureSocket(result);
    return result;
}
```

Listing B.54: Ground truth 2 for Rule 26.

B.16 RULE 29: USE RECOMMENDED CRYPTOGRAPHY ALGORITHMS

Best practice: ”Use a recommended cryptographic algorithm if dealing with Cipher, Mac, MessageDigest, Signature objects.”

```
byte[] salt = {(byte) 0xA9, (byte) 0x9B, (byte) 0xC8, (byte) 0x32, (byte) 0x56,
     (byte) 0x34, (byte) 0xE3, (byte) 0x03};
int iterationCount = 19;

try {
    KeySpec keySpec = new PBEKeySpec(passPhrase.toCharArray(), salt,
    iterationCount);
    SecretKey key = SecretKeyFactory.getInstance("PBEWithMD5AndDES").
    generateSecret(keySpec);

    ecipher = Cipher.getInstance(key.getAlgorithm());
    // rest of the code

    ecipher.init(Cipher.ENCRYPT_MODE, key, paramSpec);
    // rest of the code
}
```

Listing B.55: Vulnerable code 1 for Rule 29.

```
// fix29: use only a recommended algorithm
// choose between: "AES/CBC/NoPadding" or "AES/CBC/PKCS5Padding" or "AES/GCM/
    NoPadding"
// Cipher <cipher> = <cipher>.getInstance(<recommended_algorithm>);
```

```

byte[] salt = {(byte) 0xA9, (byte) 0x9B, (byte) 0xC8, (byte) 0x32, (byte) 0x56,
     (byte) 0x34,
(byte) 0xE3, (byte) 0x03};
int iterationCount = 19;
try
{
    KeySpec keySpec = new PBEKeySpec(passPhrase.toCharArray(), salt,
    iterationCount);
    // fix29: need to change SecretKeyFactory instance into "
    PBKDF2WithHmacSHA256" and add IV (Initialization Value)
    SecretKey key = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256").
    generateSecret(keySpec);
    byte[] iv = new byte[16];
    new SecureRandom().nextBytes(iv);
    IvParameterSpec iv_param = new IvParameterSpec(iv);

    ecipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    // rest of the code

    // fix29: need to change how to encrypt following AES protocol
    ecipher.init(Cipher.ENCRYPT_MODE, key, iv_param);
    // rest of the code
}
```

Listing B.56: Ground truth 1 for Rule 29.

```
public String getInstallationId() {
    String installation_id = (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) ?
     Settings.Secure.getString(mContext.getContentResolver(), Settings.Secure.
    ANDROID_ID) : Build.SERIAL;

    try {
        MessageDigest md5 = MessageDigest.getInstance("MD5");
        byte[] digest = md5.digest(installation_id.getBytes());
        installation_id = "M" + Utils.byteArrayToHex(digest, 8);
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
        installation_id = "D" + installation_id;
    }

    return installation_id;
```

```
}
```

Listing B.57: Vulnerable code 2 for Rule 29.

```
// fix29: use only a recommended algorithm
// MessageDigest <msg_digest> = MessageDigest.getInstance("SHA-256");

public String getInstallationId()  {
    String installation_id = (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) ?
     Settings.Secure.getString(mContext.getContentResolver(), Settings.Secure.
    ANDROID_ID) : Build.SERIAL;
    try  {
        // fix29
        MessageDigest md5 = MessageDigest.getInstance("SHA-256");
        byte[] digest = md5.digest(installation_id.getBytes());
        installation_id = "M" + Utils.byteArrayToHex(digest, 8);
    } catch (NoSuchAlgorithmException e)  {
        e.printStackTrace();
        installation_id = "D" + installation_id;
    }
    return installation_id;
}
```

Listing B.58: Ground truth 2 for Rule 29.