Enhancing Android application security through source code vulnerability mitigation using artificial intelligence: a privacy-preserved, community-driven, federated-learning-based approach.

SENANAYAKE, J.M.D.

2024

Enhancing Android Application Security through Source Code Vulnerability Mitigation using Artificial Intelligence

A Privacy-Preserved, Community-Driven, Federated-Learning-based Approach

Janaka Maduwantha Dias Senanayake

PhD 2024

Enhancing Android Application Security through Source Code Vulnerability Mitigation using Artificial Intelligence

A Privacy-Preserved, Community-Driven, Federated-Learning-based Approach

Janaka Maduwantha Dias Senanayake

A thesis submitted in partial fulfilment of the
requirements of
Robert Gordon University
for the degree of Doctor of Philosophy

October 2024

Abstract

As technology advances, Android devices and apps are rapidly increasing. It is crucial to adhere to security protocols during app development, especially as many apps lack sufficient safeguards. Despite the use of automated tools for risk mitigation, their ability to detect vulnerabilities is limited. Therefore, this doctoral research endeavours to propose a novel, highly accurate, efficient, privacy-preserved, and community-driven approach that utilises Artificial Intelligence (AI) techniques to detect Android source code vulnerabilities in real time, with a focus on continuous model improvement.

To train the initial AI model, a dataset has been curated, containing labelled Android source code based on the Common Weakness Enumeration (CWE) obtained by scanning over 15,000 real-world Android apps. A proof-of-concept has been presented, showcasing the suitability of the dataset for training various Machine Learning (ML) models. The model then evolves into a deep learning-based system incorporating a shallow neural network. Enhancing the model's performance necessitates the collection of additional data from a variety of sources. This could encompass source code from both software firms and solo developers, in addition to the LVDAndro dataset. It is crucial to respect the privacy of their code in this process. To this end, the final model integrates a federated learning method underpinned by blockchain technology, ensuring security, privacy, and community involvement. The ultimate models exhibit excellent performance, with both binary and multi-class models achieving an accuracy of 96% and an F1-Score of 0.96. The model's predictions are further clarified using Explainable AI (XAI), providing developers with guidance on potential mitigation strategies.

The AI model is designed to integrate into an API as a backend and is also integrated as a plugin in Android Studio. This setup allows for instantaneous detection of vulnerabilities, taking on average 300ms to scan a single line of code. Utilising this plugin, app developers have a way to build safer applications, thus reducing the risk of source code vulnerabilities. In addition, Android app developers have tested the solution and found the plugin to be highly effective in real-time vulnerability mitigation.

Keywords: Android, Code Vulnerabilities, Software Security, Artificial Intelligence, Explainable AI, Federated Learning, Blockchain, Android Studio Plugin

Acknowledgements

I am immensely grateful to numerous individuals whose support and assistance in conducting this research cannot be adequately expressed in a few words.

I express my profound gratitude to the School of Computing at Robert Gordon University for their unwavering support during my PhD research. Special thanks are due to my Principal supervisor, Dr. Harsha Kalutarage, for his invaluable guidance, continuous support, motivation, and exceptional supervision. I am also grateful for the constructive feedback, ideas, and support provided by my other supervisors, including Dr. MhD Omar Al-Kadri from the University of Doha for Science and Technology, Qatar, Dr. Andrei Petrovski from Robert Gordon University, and Dr. Luca Piras from Middlesex University, UK, which greatly contributed to the success of this project.

I extend my thanks to the University of Kelaniya, Sri Lanka, and the AHEAD grant of Sri Lanka for their support in facilitating this research.

Lastly, I would like to acknowledge and appreciate the understanding and support of my wife throughout this journey. The consistent support from my parents, parents-in-law, sister, sister-in-law, brothers-in-law, and all other family members played a crucial role in encouraging me to undertake and complete this successful project.

Contents

Abstract ii

Acknowledgements iii

1 Introduction 1
- 1.1 Research Motivation . . . 1
- 1.2 Research Questions . . . 4
- 1.3 Research Objectives . . . 4
- 1.4 Research Contributions . . . 6
  - 1.4.1 List of Publications . . . 7
  - 1.4.2 List of Open-source Repositories . . . 8
- 1.5 Scope and Limitations . . . 9
- 1.6 Thesis Structure . . . 11

2 Background 13
- 2.1 Security Implications of Android . . . 13
- 2.2 Vulnerabilities of Android Applications . . . 14
- 2.3 Source Code Vulnerabilities . . . 15
- 2.4 Machine Learning Processes . . . 18
- 2.5 XAI for Interpreting AI Prediction Results . . . 19
- 2.6 Data Scarcity . . . 20
- 2.7 Federated Learning and Differential Privacy for AI Models . . . 21
- 2.8 Blockchain-based Federated Learning . . . 24
- 2.9 Chapter Summary . . . 25

3 Literature Review 26
- 3.1 Conducting the Literature Review . . . 27
  - 3.1.1 Search Strategy . . . 28
  - 3.1.2 Study Selection, Data Extraction, and Synthesis . . . 30
  - 3.1.3 Threats to Validity of the Review . . . 30
- 3.2 Related Literature Reviews . . . 32
- 3.3 Application Analysis . . . 36
  - 3.3.1 Static Analysis . . . 38

3.3.2 Dynamic Analysis . . . 39
3.3.3 Hybrid Analysis . . . 39
3.4 Code Vulnerability Detection . . . 40
3.4.1 Using Machine Learning Methods . . . 40
3.4.2 Using Conventional Methods . . . 50
3.4.3 Prevention Techniques . . . 58
3.5 Use of XAI . . . 60
3.6 Existing Tools for Analysing Apps and Detecting Vulnerabilities . . . 61
3.7 Repositories and Datasets for Vulnerability Detection . . . 68
3.8 Discussion on Application Analysis and Vulnerability Detection Methods . . . 70
3.9 Chapter Summary . . . 73

4 Methodology **74**
4.1 Overall Approach . . . 74
4.2 Developing the Android Code Vulnerability Dataset . . . 75
4.3 Developing AI-based Android Code Vulnerability Detection Model . . . 78
4.4 Enhancing the Model to a Privacy-Preserved Community Driven Model . . . 80
4.5 Chapter Summary . . . 82

5 Labelled Vulnerability Dataset Generation **83**
5.1 Dateset Generation Process . . . 83
5.1.1 Scrapping APKs and Source Files (Data Collection) . . . 84
5.1.2 Scanning APKs for Vulnerabilities (Data Labelling) . . . 84
5.1.3 Creating Processed Dataset (Preprocessing) . . . 86
5.2 Resulting Dataset . . . 86
5.2.1 Different Datasets . . . 86
5.2.2 Statistics of Datasets . . . 88
5.3 Chapter Summary . . . 92

6 Vulnerability Detection using AI-based Model **93**
6.1 Proof-of-Concept Demonstration with AutoML . . . 93
6.1.1 Training AutoML Models . . . 93
6.1.2 AutoML Model Comparison . . . 95
6.2 Improving the Capabilities with Ensemble model . . . 96
6.3 Development of Neural Network-Based Vanilla Models . . . 99
6.3.1 Fine-Tuning and Pruning of Vanilla Model Parameters . . . 100
6.3.2 Performances of the Vanilla Models . . . 101
6.4 Incorporating Explainable AI with the AI-based Model and API . . . 103
6.5 Chapter Summary . . . 106

7 Privacy-preserved Community Driven Model Enhancement **107**
7.1 Training the Model in a Federated Environment . . . 108
7.1.1 Performance of the Federated Models . . . 109
7.2 Extending to a Blockchain-based Federated Environment . . . 111

7.3 Integrating Differential Privacy . . . 114
7.3.1 Applying Differential Privacy to all Clients . . . 115
7.3.2 Applying Differential Privacy to Randomly Selected Clients . . . 115
7.3.3 Applying Differential Privacy to All Clients Except Alpha . . . 116
7.3.4 Variation of Accuracy and Privacy Budget with Noise Multiplier . 118
7.3.5 Variation of Accuracy and Privacy Budget with L2_Norm_Clip . 119
7.3.6 Differential Privacy Incorporating Variable Noise and L2_Norm_Clip with Blockchain-based Federated Learning . . . 122
7.4 Application of AI-based Model using an Android Studio Plugin . . . 127
7.4.1 Plugin Integration to Android Studio . . . 128
7.4.2 Plugin Usage . . . 129
7.4.3 Assessing the Plugin Capabilities . . . 133
7.5 Chapter Summary . . . 139

8 Case Study: Use of the Defendroid Plugin for Android Code Vulnerability Detection 140
8.1 Need Analysis for Android Code Vulnerability Detection Method . . . 140
8.1.1 Characteristics of the Selected Android Developers . . . 141
8.1.2 Secure Coding Practices of the Selected Android Developers . . . 143
8.2 Developer Feedback . . . 145
8.3 Chapter Summary . . . 148

9 Conclusion 149
9.1 Discussion and Summary . . . 149
9.2 Revisiting Objectives . . . 152
9.3 Limitations . . . 154
9.4 Possible Future Directions . . . 155
9.4.1 Increase the Detection Capabilities of the Model . . . 155
9.4.2 Customise the Model to be Trained Using a Public Blockchain While Providing Attractive Rewards . . . 155
9.4.3 Complex Vulnerability Patterns Detection . . . 156
9.4.4 Increase the Model Privacy and Security more with Homomorphic Encryption and Secure Multi-Party Computation . . . 156

Bibliography 157

A Survey on Identifying the Characteristics of Selected Android Developers 176

B Developer Feedback on the Plugin 179

List of Tables

2.1 Common Vulnerabilities in Android Code 17

3.1 Distribution of Search Results from Primary Sources from 2016 to 2024 . 29
3.2 Summary of Related Reviews 37
3.3 ML/DL-based Android Vulnerability Detection Mechanisms 46
3.4 Conventional Methods of Android Vulnerability Detection 55
3.5 Existing Tools for Analysing Apps and Detecting Vulnerabilities 63

5.1 Overview of the LVDAndro datasets 87
5.2 Fields in LVDAndro 89
5.3 Available CWE-IDs in LVDAndro 90

6.1 Performance comparison of AutoML models in binary classification 94
6.2 Performance comparison of AutoML models in multi-class classification . . 95
6.3 Accuracy comparison of proposed ML model with MobSF and Qark . . . 96
6.4 Statistics of the LVDAndro Dataset 96
6.5 Performance Comparison of Learning Models 98
6.6 F1-Score for each CWE-ID with Ensemble Model 99
6.7 Performance Comparison of Vanilla Models 101

7.1 Statistics of the Client Datasets 109
7.2 Comparison of Federated Models with Vanilla Model 110
7.3 Federated Learning with Differential Privacy Applied to All Clients 115
7.4 Federated Learning with Differential Privacy Applied to Randomly Selected Clients 116
7.5 Federated Learning with Differential Privacy Applied to All Clients Except Alpha 117
7.6 Variation of Accuracy and Privacy Budget with Noise Multiplier - Binary Classification 118
7.7 Variation of Accuracy and Privacy Budget with Noise Multiplier - Multiclass Classification 119
7.8 Variation of Accuracy and Privacy Budget with L2_Norm_Clip - Binary Classification 120

7.9 Variation of Accuracy and Privacy Budget with L2_Norm_Clip - Multi-class Classification . . . 121
7.10 Statistics of New Clients' Data . . . 124
7.11 Evaluation of DPV-BLFedAvg Algorithm Results - Binary Classification . 125
7.12 Evaluation of DPV-BLFedAvg Algorithm Results - Multi-class Classification 126
7.13 Accuracy Comparison of MobSF and Qark with Defendroid . . . 134
7.14 Comparison of Average Time Taken to Analyse apps . . . 135
7.15 Key Features Comparison of Defendroid with Other Popular Vulnerability Detection Tools . . . 137
7.16 Features Summary of Defendroid with Other Popular Vulnerability Detection Tools . . . 138

List of Figures

1.1 Mapping of Research Questions, Objectives, Contributions and Knowledge Dissemination . . . 10

3.1 PRISMA method: collection of papers for the review . . . 31
3.2 Application/source code analysis techniques used in the reviewed studies . . . 70
3.3 Vulnerability detection methods . . . 71
3.4 Feature extraction methods used in the reviewed studies . . . 71
3.5 Extracted features in the reviewed studies . . . 72
3.6 Availability of detection and prevention methods . . . 73

4.1 Overall Approach . . . 76
4.2 The process of generating the LVDAndro dataset . . . 76
4.3 Overview of LVDAndro Datasets . . . 77
4.4 API Development Process . . . 79
4.5 Overview of the FedREVAN Model . . . 80
4.6 Environment Changes from FedREVAN to Defendroid . . . 82
4.7 Process of Vulnerability Detection Plugin . . . 82

5.1 Origins of the downloaded applications . . . 84
5.2 Distribution of vulnerable and non-vulnerable code samples in each dataset (using the APK Combined Approach) . . . 88
5.3 Distribution of CWE-IDs in dataset 03 . . . 91
5.4 Spread of CWE-IDs according to the likelihood of exploitation . . . 91

6.1 Accuracy and Loss with Epochs - Vanilla Models . . . 102
6.2 Example API Responses for Given Codes . . . 104
6.3 Example XAI Representations for Given Codes . . . 105

7.1 Federated Learning Simulated Environment . . . 108
7.2 Accuracy and Loss with Epochs - Defendroid Models . . . 113
7.3 Variation of Accuracy and Privacy Budget with Noise Multiplier - Binary Classification . . . 119
7.4 Variation of Accuracy and Privacy Budget with Noise Multiplier - Multi-class Classification . . . 120

7.5 Variation of Accuracy and Privacy Budget with L2_Norm_Clip - Binary Classification . . . 121
7.6 Variation of Accuracy and Privacy Budget with L2_Norm_Clip - Multi-class Classification . . . 122
7.7 Plugin Integration with Android Studio in Tools Menu . . . 129
7.8 Quick Check Notification of Non-vulnerable Source File in Android Studio 130
7.9 Quick Check Notifications - Vulnerable Source File . . . 131
7.10 Detail Check Notification for a Non-Vulnerable Code Line in Android Studio 132
7.11 Detail Check Notifications - Vulnerable Source File . . . 133
7.12 Plugin Notifications in Event Log . . . 134

8.1 Characteristics Analysis Q1 to Q4 . . . 141
8.2 Characteristics Analysis Q5 to Q10 . . . 142
8.3 Consideration of Secure Coding . . . 143
8.4 Reasons for Underestimating Secure Coding . . . 144
8.5 Developer Satisfaction . . . 146
8.6 Overall Satisfaction of the Plugin . . . 147

List of Algorithms

1 Federated Averaging (FedAvg) Algorithm 22
2 Differential Privacy Algorithm using TensorFlow Privacy 23

3 Consensus Algorithm 112
4 Blockchain-based Federated Averaging with Differential Privacy incorporating Variable Noise and L2_Norm_Clip (DPV-BLFedAvg) Algorithm . . 123

Chapter 1

Introduction

In the realm of Android application development, it is crucial to promptly identify and rectify vulnerabilities in the source code. Starting this vital process in the early stages of development is particularly important, as it greatly minimises the chances of attackers finding and exploiting these vulnerabilities [1]. As of February 2024, Android holds a significant market share of 71.4% and sees an influx of approximately 52,000 new mobile apps on the Google Play Store each month, making it a widely used platform [2]. However, unlike iOS applications, Android apps often lack thorough security assessments [3], highlighting the necessity to modify the development process to meet stringent security standards for Android apps.

Despite thorough requirements analysis and feasibility studies at the outset of development, the end product may still be prone to failure due to vulnerabilities in the code. It is important to note that rectifying bugs early in the Software Development Life Cycle (SDLC) is about 70 times more cost-effective than fixing them in the later stages [4]. As a result, researchers have developed a variety of automated tools to detect vulnerabilities in Android apps [5]. These tools aim to prioritise security-centric development and proactively avert cybersecurity breaches, rather than dealing with issues later in the app development life cycle.

1.1 Research Motivation

In previous studies, the development of numerous tools, frameworks, and plugins aimed at assisting developers in automating the vulnerability detection process was considered

[6]. These tools leverage both conventional techniques and advanced methods rooted in Machine Learning and Deep Learning to spot vulnerabilities in Android applications.

These techniques use static, dynamic, and hybrid analysis methods to scrutinise either the Android Application Package (APK) files or the entire Android project source files for vulnerability detection. Static analysis evaluates an application's code without executing it, making it particularly effective at identifying potential issues early on. This method can detect vulnerabilities during development, even before deployment, which is especially beneficial for resource-constrained Android devices. However, static analysis may lack precision due to its inability to fully comprehend dynamic behaviour. It also faces challenges with scalability when analysing large codebases and can be evaded by repackaged, polymorphic, or code-transformed malware apps. In contrast, dynamic analysis examines program behaviour during runtime, providing high precision and better code coverage by actually executing the app. This approach overcomes the limitations posed by obfuscated code. However, dynamic analysis incurs runtime overhead due to execution monitoring, and vulnerabilities are only detected during runtime, which may be too late for prevention. Additionally, it may not cover all execution paths, particularly if certain conditions are not met during testing. Hybrid analysis integrates both static and dynamic approaches. Nevertheless, for detecting code vulnerabilities at the early stages, static analysis is considered the most appropriate method [7, 8].

However, a notable drawback of these current solutions is their inability to facilitate early detection of vulnerabilities in a real-time app development setting. These tools are only capable of identifying vulnerabilities by scanning the code after the development process is complete.

The integration of AI into source code vulnerability detection offers several compelling reasons. Firstly, AI algorithms efficiently identify vulnerabilities in source code, reducing dependence on human expertise and automating the detection process. By utilising semantic code analysis engines, AI-powered tools proactively discover security issues before they are merged into the codebase and released. Additionally, AI models provide real-time predictions and suggest fixes for detected vulnerabilities, thereby enhancing the overall security posture of software projects. Given that programming languages share similarities with natural languages—comprising words, numbers, and symbols—AI's capacity to process and analyse code at scale becomes indispensable for enforcing robust security practices in software development [9, 10, 11].

Utilising AI-based methods on a properly annotated dataset of Android source code vulnerabilities can efficiently overcome these limitations. However, it is crucial to recognise the limitations tied to the datasets used for training models to detect Android vulnerabilities. One possible approach is to build a dataset by labelling the source code after examining released APKs, but this approach has its drawbacks. The scope of the dataset, including the count of unique vulnerability categories, is limited, and it might not have enough code examples of new vulnerabilities. Another approach is to train a model using source code obtained from app developers. However, due to privacy concerns, developers may be reluctant to share their proprietary code [12, 13].

To overcome these limitations in the model training process, federated learning can be utilised [14]. This method distributes the model training process across multiple entities within a federated network. As a result, these entities can independently train the model and contribute improvements to the final model without exposing their data, which includes source code samples. However, a limitation of the current federated learning approach is its restricted capacity to involve and motivate the participating clients in cooperative training to enhance model performance while ensuring its accuracy. While federated learning can secure the data, the application of differential privacy can enhance the privacy of the model. Therefore, a blockchain-based federated learning approach with the application of differential privacy can be adopted to address this. In this approach, model weights are shared within the blockchain, and new model updates act as incentives for those who genuinely contribute to improving the detection capabilities of the model. Therefore, this research seeks to explore the application of AI-based techniques to propose and construct a privacy-focused, highly accurate, community-driven method.

The use of XAI is also crucial as it can elucidate the rationale behind the vulnerability predictions made by the AI-based model [15]. By leveraging these insights, developers can take proactive measures to rectify code vulnerabilities, thereby enhancing the security of their applications. This process becomes even more streamlined and efficient when all these elements are incorporated into an Android Studio plugin. By integrating the AI model into a plugin for this platform, developers can have real-time access to vulnerability detection and their explanations right within their development environment. This allows them to mitigate vulnerabilities as they code, significantly reducing the time and effort typically required for vulnerability detection and mitigation.

Therefore, this doctoral research proposal holds significant potential for contributing to the development of secure mobile applications. By minimising source code vulnerabilities, it can help create a safer digital environment for users and developers alike. This approach aligns with the broader goal of enhancing cybersecurity in the era of digital transformation, where secure coding practices are of paramount importance. Thus, the proposed research could be a valuable addition to the field of secure mobile application development.

The research aims to develop a robust, privacy-preserving, and blockchain-integrated AI-based framework for the real-time detection and mitigation of Android code vulnerabilities. This will be achieved by leveraging differential privacy and federated learning to enhance community contributions and ensure distributed ownership and audit of updates.

1.2 Research Questions

With the above research motivation, the following research questions (RQ) were formulated.

RQ1: What are the current methods based on static, dynamic, and hybrid analysis for identifying vulnerabilities in the source code of mobile applications?

RQ2: What are the available datasets related to Android code vulnerabilities, their limitations, and what methodologies could be employed to generate such datasets using the vulnerability scanners and analysis tools identified in RQ1?

RQ3: How to develop a highly precise AI-based model for detecting vulnerabilities using the dataset created in RQ2?

RQ4: How to improve the capabilities of the model developed in RQ3 while ensuring the security and privacy of the training data?

1.3 Research Objectives

Aligning with the aim of the research the project involves conducting an extensive literature survey to evaluate current vulnerability detection methods, generating a novel labelled dataset, developing a highly accurate neural network-based detection model, devising strategies for mitigating identified vulnerabilities, and enhancing the model's detection capabilities through privacy-preserving data improvement techniques.

The aforementioned research questions were addressed in the process of accomplishing the following research objectives (RO), while setting the scope of the research around them.

RO1: To recognise and evaluate current methods for detecting vulnerabilities in Android source code through an extensive literature survey.

In the process of accomplishing this objective, RQ1 can also be addressed This would necessitate conducting a comprehensive review of the literature. Through this review, the existing methods for detecting code vulnerabilities, as well as their capabilities, limitations, and applications can be identified.

RO2: To generate a labelled dataset of vulnerabilities in Android source code.

In the process of addressing RQ2, the existing datasets, their capabilities, and their limitations can be investigated. This exploration can also help identify the need for a new dataset. Consequently, a novel labelled dataset can be created that addresses these limitations, which is a crucial step in achieving this objective.

RO3: To develop an accurate AI-based technique for real-time detection of source code vulnerabilities in Android app development, utilising the dataset generated from RO2.

While tackling RQ3, it is necessary to develop a highly accurate AI-based method. This method should be trained using the appropriately labelled dataset created during the resolution of RO2. The procedure for developing such a dataset is examined as part of achieving this objective.

RO4: To devise a strategy for offering suggestions to mitigate code vulnerabilities, utilising the method developed in RO3.

Merely providing prediction results is not enough to assist developers in addressing the identified vulnerabilities. Understanding the rationale behind these predictions can also be beneficial. Therefore, by using the AI-based model developed in RO3, a method for proposing potential mitigation strategies is offered as part of this objective. This approach further enriches the response to RQ3.

RO5: To improve the training data while preserving privacy and enhancing the detection capabilities of the model developed in RO3.

In the process of addressing RQ4, it is crucial to explore a method for enhancing

the model by improving the training data. Training an AI model with a diverse dataset is generally more effective. However, the scarcity of data, particularly due to proprietary source code, presents a challenge. Therefore, integrating a privacy-preserving method that encourages more community engagement is an essential part of achieving this objective.

1.4 Research Contributions

This thesis primarily presents a community-driven privacy-preserved method for detecting vulnerabilities in Android source code in real-time with high accuracy, leveraging an AI-based backend. This system also supports model training driven by the community. The below list summarises the key research contributions (RC) of this research:

RC1: The initial offering of this thesis is an exhaustive and methodical review of the detection of vulnerabilities in Android source code, as detailed in Chapter 3. This review pinpoints the areas of research that have been overlooked and examines how current studies attempt to tackle the complex issues associated with detecting vulnerabilities while achieving RO1 and addressing RQ1. The review significantly influences the research community by meticulously and systematically examining high-quality technical papers, ensuring all the crucial aspects are covered.

RC2: The second key element of this thesis is the creation of a CWE-labelled dataset for Android source code vulnerabilities, referred to as LVDAndro, as elaborated in Chapter 5. This newly proposed dataset can serve as a foundation for constructing AI models to identify vulnerabilities in the Android application's source code. RO2 was achieved in this while addressing RQ2. Considering previous studies, the LVDAndro dataset is unique as it is the only publicly available dataset that contains properly labelled Android source code, with vulnerabilities annotated based on CWE-IDs.

RC3: The third contribution of this thesis is developing a precise AI model for detecting vulnerabilities in Android's source code which relates to addressing RQ3 while achieving RO3 and RO4. The construction of this AI model, which incorporates ML, AutoML, DL models, and XAI, is facilitated by the use of the LVDAndro dataset and is discussed in Chapter 6. The experiments involving the development of an AI-based model and its reasoning abilities underscored the importance of the model development approach that utilised the newly formed dataset.

RC4: The fourth contribution of this thesis was the improvement of the model's performance by leveraging more training data to meet RO4 and address RQ4, while keeping the source code's privacy intact. This was accomplished by applying federated learning in conjunction with differential privacy, and by attracting more clients through a community-driven blockchain network. This topic is explored in Chapter 7. Furthermore, this model has been incorporated as a plugin in Android Studio, serving as a helpful tool for app developers to effectively mitigate vulnerabilities in real-time. The applicability of this plugin has been confirmed through testing with a group of Android developers. This thesis's most notable contribution is the proposed innovative method based on a privacy-preserving, community-driven federated learning approach for real-time detection of Android code vulnerabilities, employing a plugin that Android developers can utilise.

1.4.1 List of Publications

Following is the list of publications that have been produced as knowledge dissemination (KD) during the course of the research presented in this thesis:

- Journals:

KD1: Senanayake, J., Kalutarage, H. and Al-Kadri, M.O., 2021. Android mobile malware detection using machine learning: A systematic review. In Special Issue High Accuracy Detection of Mobile Malware Using Machine Learning on Electronics, 10(13), p.1606. https://doi.org/10.3390/electronics10131606 [16].

KD2: Senanayake, J., Kalutarage, H., Al-Kadri, M.O., Petrovski, A. and Piras, L., 2023. Android source code vulnerability detection: a systematic literature review. ACM Computing Surveys, 55(9), pp.1-37. https://dl.acm.org/doi/full/10.1145/3556974 [17].

KD7: Senanayake, J., Kalutarage, H., Petrovski, A., Piras, L., and Al-Kadri, M. O. (2024). Defendroid: Real-time Android code vulnerability detection via blockchain federated neural network with XAI. Journal of Information Security and Applications, 82, 103741. https://doi.org/10.1016/J.JISA.2024.103741 [18].

- Conferences:

KD3: Senanayake, J.; Kalutarage, H.; Al-Kadri, M.; Piras, L. and Petrovski, A. (2023). Labelled Vulnerability Dataset on Android Source Code (LVDAndro) to Develop AI-Based Code Vulnerability Detection Models. In Proceedings of the 20th International Conference on Security and Cryptography - SECRYPT; ISBN 978-989-758-666-8; ISSN 2184-7711, SciTePress, pages 659-666. https://doi.org/10.5220/0012060400003555 [19].

KD4: Senanayake, J., Kalutarage, H., Al-Kadri, M.O., Petrovski, A. and Piras, L., 2022, May. Developing secured android applications by mitigating code vulnerabilities with machine learning. In Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security (ASIACCS) (pp. 1255-1257). https://dl.acm.org/doi/abs/10.1145/3488932.3527290 [20].

KD5: Senanayake, J., Kalutarage, H., Al-Kadri, M.O., Petrovski, A. and Piras, L., 2023, July. Android Code Vulnerabilities Early Detection Using AI-Powered ACVED Plugin. In IFIP Annual Conference on Data and Applications Security and Privacy (DBSec) (pp. 339-357). Cham: Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-37586-6_20 [21].

KD6: Senanayake, J., Kalutarage, H., Petrovski, A., Al-Kadri, M.O., Piras, L. (2024). FedREVAN: Real-time DEtection of Vulnerable Android Source Code Through Federated Neural Network with XAI. In Computer Security. ESORICS 2023 International Workshops. ESORICS 2023. Lecture Notes in Computer Science, vol 14399. Springer, Cham. https://doi.org/10.1007/978-3-031-54129-2_25 [22]

KD8: Senanayake, J., Kalutarage, H., Piras, L., Al-Kadri, M.O. and Petrovski, A., (2024). Assuring Privacy of AI-Powered Community Driven Android Code Vulnerability Detection. In Computer Security. ESORICS 2024 International Workshops. ESORICS 2024. Lecture Notes in Computer Science. Springer, Cham. [23]

1.4.2 List of Open-source Repositories

Following is the list of open-source code repositories (CR) and the dataset (DS) which have been produced during the course of the research presented in this thesis:

CR1: Defendroid - https://github.com/softwaresec-labs/Defendroid

This repository contains the source code and details of the final model and the plugin integrating the community-driven privacy-preserved federated learning-based AI techniques, integrating XAI for Android code vulnerability detection.

CR2: FedREVAN - https://github.com/softwaresec-labs/FedREVAN

This repository contains the source code and details of the federated learning-based AI model, integrating XAI for Android code vulnerability detection.

CR3: ACVED - https://github.com/softwaresec-labs/ACVED

This repository contains the source code and details of the ensemble model, integrating XAI for Android code vulnerability detection.

DS1: LVDAndro - https://github.com/softwaresec-labs/LVDAndro

This dataset contains the CWE-based labelled data of Android code vulnerabilities.

The mapping between the research questions (RQ), objectives (RO), contributions (RC), and knowledge dissemination (KD) is illustrated in Figure 1.1.

1.5 Scope and Limitations

This doctoral research is focused on the detection of vulnerabilities in Android source code using an AI-based method that leverages static analysis techniques. The goal of the research is to devise an innovative and precise strategy that considers the privacy of the source code and the support of the community to identify vulnerabilities in Android code in real-time. The model developed as part of this research and its associated plugin can be utilised by Android developers during the coding phase of their applications. This allows for the early detection and rectification of potential vulnerabilities, thereby enhancing the overall security of the apps. However, it is important to note that the model's detection capabilities are confined to a static code analysis approach. This means that it will not be able to detect dynamic or logical vulnerabilities that could potentially arise during the runtime of the application.

In addition to the development of the AI model, the research also explored the use of blockchain-based federated learning integrating differential privacy. This was done in response to infrastructure constraints that often pose challenges in the field of AI. The

Research Questions		Research Objectives		Research Contribution		Knowledge Dissemination	
RQ1	What are the current methods based on static, dynamic, and hybrid analysis for identifying vulnerabilities in the source code of mobile applications?	RO1	To recognise and evaluate current methods for detecting vulnerabilities in Android source code through an extensive literature survey.	RC1	An exhaustive and methodical review of the detection of vulnerabilities in Android source code, pinpointing the areas of research that have been overlooked and examining how current studies attempt to tackle the complex issues associated with detecting vulnerabilities.	KD1	Android mobile malware detection using machine learning: A systematic review. In Special Issue High Accuracy Detection of Mobile Malware Using Machine Learning on Electronics. 2021
						KD2	Android source code vulnerability detection: a systematic literature review. ACM Computing Surveys. 2023
RQ2	What are the available datasets related to Android code vulnerabilities, their imitations, and what methodologies could be employed to generate such datasets using the vulnerability scanners and analysis tools identified in RQ1?	RO2	To generate a labelled dataset of vulnerabilities in Android source code.	RC2	Creation of a CWE-labelled dataset for Android source code vulnerabilities, named LVDAndro, which can serve as a foundation for constructing AI models to identify vulnerabilities in the Android application's source code.	KD3	Labelled Vulnerability Dataset on Android Source Code (LVDAndro) to Develop AI-Based Code Vulnerability Detection Models. In Proceedings of the 20th International Conference on Security and Cryptography – SECRYPT. 2023
RQ3	How to develop a highly precise AI-based model for detecting vulnerabilities using the dataset created in RQ2?	RO3	To develop an accurate AI-based technique for real-time detection of source code vulnerabilities in Android app development, utilising the dataset generated from RO2.	RC3	Development of a precise AI model for detecting vulnerabilities in Android's source code. The construction of this AI model, which incorporates ML, AutoML, DL models, and XAI, is facilitated by the use of the LVDAndro dataset. Incorporation of the model as a plugin in Android Studio serves as a helpful tool for app developers to effectively mitigate vulnerabilities in real-time.	KD4	Developing secured Android applications by mitigating code vulnerabilities with machine learning. In Proceedings of the ACM on Asia Conference on Computer and Communications Security (ASIACCS). 2022
		RO4	To devise a strategy for offering suggestions to mitigate code vulnerabilities, utilising the method developed in RO3.			KD5	Android Code Vulnerabilities Early Detection Using AI-Powered ACVED Plugin. In IFIP Annual Conference on Data and Applications Security and Privacy (DBSec). 2023
RQ4	How to improve the capabilities of the model developed in RQ3 while ensuring the security and privacy of the training data?	RO5	To improve the training data while preserving privacy and enhancing the detection capabilities of the model developed in RO3.	RC4	Enhancement of the model's capabilities with more training data while maintaining the privacy of the source code by applying federated learning with differential privacy and attracting more clients through a community-driven blockchain network.	KD6	FedREVAN: Real-time Detection of Vulnerable Android Source Code through Federated Neural Network with XAI. In Proceedings of the Workshop on Attacks and Software Protection (WASP) in 28th European Symposium on Research in Computer Security (ESORICS). 2023
						KD7	Defendroid: A Real-time Android Code Vulnerability Detection Approach via Blockchain Federated Neural Network with XAI. In Special Issue on Software Protection and Attacks in Journal of Information Security and Applications. 2024
						KD8	Assuring Privacy of AI-Powered Community Driven Android Code Vulnerability Detection. In Proceedings of the Workshop on System Security Assurance (SecAssure) in 29th European Symposium on Research in Computer Security (ESORICS). 2024

Figure 1.1: Mapping of Research Questions, Objectives, Contributions and Knowledge Dissemination

performance of the blockchain-based federated learning approach was tested in a simulated environment. The results demonstrated that the model performed well, indicating the potential effectiveness of this approach in a real-world setting. However, when scaling the environment to cater to a larger audience, it was noted that there may be a need to upgrade the infrastructure capabilities to a higher level. This would ensure that the system can effectively handle the increased load and continue to deliver accurate and timely vulnerability detection.

1.6 Thesis Structure

The remaining structure of this thesis is as follows:

Chapter 2 offers an introduction to the key concepts used in this work, including Android security implications, code vulnerabilities, the machine learning process, the application of XAI, and blockchain-based federated learning with differential privacy. The information provided in the background is beneficial for comprehending the technical aspects discussed in subsequent chapters.

Chapter 3 delves into the existing work on detecting vulnerabilities in Android's source code. It covers various analysis methods, both ML-based and non-ML-based approaches to vulnerability detection, available tools and datasets, and the security and privacy aspects of sharing source code for training AI models. The findings from this chapter guided the design of the methodology and experiments, as elaborated in the following chapters.

Chapter 4 outlines the research methodology employed, detailing the various stages involved in the development of the AI model. This chapter provides a summary of the methodology employed in the subsequent chapters, which encompasses the creation of the dataset, the development of the AI model, and its enhancements with the privacy-preserved community-driven approach.

Chapter 5 presents the LVDAndro dataset, which pertains to vulnerabilities in Android's source code. It discusses the process of generating the dataset and experiments, its applications, and potential ways to expand it. The LVDAndro dataset is employed in the experiments outlined in the next chapter for the training of various AI-oriented models.

Chapter 6 explores the development of the AI model for real-time vulnerability detection. It includes proof-of-concept demonstration, and various experiments with auto ML models, ensemble models, and DL models. The use of XAI is also discussed which is used to provide reasons for AI-based predictions.The AI models, which are developed based on the experiments conducted in this chapter, serve as the foundational models for further enhancements, as detailed in the subsequent chapter.

Chapter 7 discusses how to increase the training data by involving a large number of clients who can provide source code. It addresses the security and privacy concerns of the model and the training data, using a blockchain-based federated learning architecture with differential privacy. It is followed by the construction and validation of a prototype plugin that can be integrated with Android Studio. The evaluation of the enhanced privacy-focused, community-driven model based on federated learning and its plugin usage is carried out, and the results are discussed in the following chapter with the assistance of Android app developers.

Chapter 8 discusses the case study based on developer feedback on Android code vulnerability mitigation and how such an automated tool as proposed in this work can be utilised. The final chapter discusses a number of potential enhancement areas, as identified from the developer feedback highlighted in this chapter, along with additional recommendations for further refining the model.

Chapter 9 discusses the research findings and concludes the thesis. It summarises the contributions and key outcomes of this research and discusses potential directions for future research to highlight possible improvements.

Chapter 2

Background

This chapter lays the background for the entire study, delving into the associated security implications of Android, vulnerabilities of Android applications, and source code vulnerabilities. It also explores the process of machine learning, the application of XAI for interpreting AI prediction results, and the concepts of blockchain, federated learning, and differential privacy.

2.1 Security Implications of Android

Mobile devices, due to their portability, are prone to being lost or stolen, frequently connect to various networks, and often contain a significant amount of privacy-sensitive data as they are typically in close proximity to the users [24]. As such, relying solely on traditional security mechanisms may not provide adequate protection for mobile devices. Potential threats include unauthorised physical access to the device, connection to untrusted networks, installation and operation of untrusted applications, and execution of unverified code blocks and content [25]. These threats are particularly relevant to Android mobile devices. Therefore, it is crucial to enhance security measures to protect the data stored on Android devices.

Android possesses a layered structure that facilitates systematic communication with device components, software applications, and users. The Android Operating System (OS) is constructed on the foundation of the Linux kernel [26], which provides drivers and mechanisms for networking, manages virtual memory, device power, and ensures security. Above the Linux kernel layer, there are several other layers: the hardware

abstraction layer, the native C/C++ libraries layer, the Android runtime layer, and the Java Application Programming Interface (API) framework layer [27].

Each of these layers carries out distinct functions while interacting with the other layers. The study in [26] proposed a layered methodology for Android application development that utilises this layered architecture. In this methodology, a server communicates with the Hypertext Transfer Protocol (HTTP) layer, and the API layer interacts with both the HTTP layer and the generic data layer. This generic data layer then interacts with the platform-dependent data layer, which in turn interacts with the User Interface (UI) layer. A significant number of source code vulnerabilities can be detected in the top layer, which houses user and system apps, as this is the layer that regular app developers primarily focus on. Nonetheless, a comprehensive understanding of the layered architecture can be instrumental in mitigating some of these vulnerabilities.

The security of the Android platform is governed by several rules outlined in the Android security model [25, 28]. These rules include multi-party consent, open ecosystem access, security and compatibility requirements, the ability to restore the device to a safe state via factory reset, and principles for application security. The study in [29] highlighted three primary security mechanisms: 1) the process sandbox, which is Android's sandbox environment; 2) the signature mechanism, which allows applications to be digitally signed with a private key before release; and 3) the permission mechanism, which determines an app's ability to access protected APIs and resources. The sandbox environment of Android restricts one application's use of another application's resources. Sandboxes, which are built using Linux, are the only entities that can access the core functionalities of the OS. The Sandbox is responsible for monitoring and acknowledging system calls [26], and it serves to thwart malicious applications that attempt to access overall system functionalities through vulnerable source code.

2.2 Vulnerabilities of Android Applications

A large number of Android applications are freely accessible for download from app markets, leading to their widespread use. If these applications lack adequate security mechanisms, hackers could potentially infiltrate them and extract user data on a large scale or engage in illegal activities [30]. As such, it is crucial for app developers to ensure the implementation of proper security measures. Common sources of vulnerabilities in Android mobile apps include issues with Secure Sockets Layer (SSL), Transport Layer Security (TLS) commands, permissions, web views, key stores, fragments, encryptions,

intents, intent filters, and leaks [31]. These vulnerabilities can be exploited by attacks from the Internet and Wireless Personal Area Networks (WPAN), as well as malware transmitted via personal computers. A study on Android vulnerability [29] identified SSL/TLS protocol vulnerabilities, forged signature vulnerabilities, and common vulnerabilities in Input/ Output (I/O) operations, intents, permissions, and web views.

CyBOK [32] can be used to understand some of the vulnerabilities associated with Android applications. It categorises several types of vulnerabilities relevant to mobile apps, including memory management vulnerabilities, structured output generation vulnerabilities, race condition vulnerabilities, API vulnerabilities, and side-channel vulnerabilities. Memory management vulnerabilities encompass safe languages, spatial vulnerabilities, temporal vulnerabilities, code corruption attacks, control-flow hijack attacks, information leak attacks, and data-only attacks. Structured output generation vulnerabilities include Structured Query Language (SQL) injections, command injection vulnerabilities, script injection vulnerabilities, stored injection vulnerabilities, and high-order injection vulnerabilities. Race condition vulnerabilities are characterised by concurrency bugs and time-of-check to time-of-use issues. API vulnerabilities can arise from incorrect use and implementation, while side-channel vulnerabilities cover software-based side channels, covert channels, micro-architectural effects, and fault injection attacks.

The research in [33] pinpointed 563 vulnerabilities related to Android, encompassing privilege and information gain, memory corruption, Denial of Services (DoS), malicious code execution, overflow, and security measure bypass. Additionally, it analysed the trends of these vulnerabilities from 2009 to 2019, noting that the peak period for vulnerabilities began in 2016. An empirical study carried out in [34] examined the types of vulnerabilities related to Android, the Android layers and subsystems that could be impacted by these vulnerabilities, and the survivability of these vulnerabilities. This study incorporated 660 vulnerabilities from the Common Vulnerabilities and Exposures (CVE) Details [35] and the official Android Security Bulletins [36]. It was determined that most vulnerabilities could arise from issues with data processing, access controls, memory buffers, and improper input validation, primarily due to lines of vulnerable source code.

2.3 Source Code Vulnerabilities

During the development of applications, developers can make mistakes and may not adhere to a thorough testing and validation process from the early stages of the app development life cycle. Despite the availability of mechanisms for writing secure codes, many

mobile app developers do not prioritise this aspect [37]. As a result of these oversights, vulnerabilities can arise from source code. Additionally, issues in the AndroidManifest.xml file can lead to vulnerabilities in mobile applications. This file provides essential information to the Android operating system, identifying the app through its package name and declaring all components, such as activities, services, broadcast receivers, and content providers. It also specifies the app's metadata, including version numbers and minimum/target API levels. One common mistake developers make is including unnecessary permissions, which can expose the app to security risks. At times, app users grant permissions without fully understanding their implications when installing or running an application, which can also lead to vulnerabilities. If these permissions are of a dangerous level and require user approval, they can result in some users rejecting the app [38]. These vulnerabilities might go undetected when apps are published to Google Play, as Google Play does not perform a comprehensive analysis of the mobile applications' code upon publishing, unlike the Apple App Store [39]. The authors in [40] found that most security issues in mobile applications are due to user actions. Therefore, it is crucial to incorporate proper mechanisms for detecting source code vulnerabilities into the coding environment.

Source code vulnerabilities include unintentional mistakes, design deficiencies, or oversights in the code. These can be exploited by malicious entities to undermine the security or functionality of the software. Examples of such vulnerabilities include buffer overflows, insecure authentication, and authorisation, issues with deserialisation, security misconfigurations, and injection vulnerabilities [41]. These vulnerabilities are relevant to Android and can be categorised using the CWE categorise, which serves as a useful reference for classifying vulnerable source code. For a more in-depth understanding of these vulnerabilities, developers can refer to the Mitre CWE repository [42]. The CWE repository aims to standardise the identification and description of vulnerabilities, facilitating better understanding and communication of these issues within the cybersecurity community. CWE enables consistent communication among developers, security professionals, and tools by providing a common language for describing software and hardware vulnerabilities. The repository includes a catalogue of common software and hardware weaknesses, each identified by a unique CWE identifier. These weaknesses cover a wide range of issues, such as buffer overflows, cross-site scripting (XSS), and improper authentication. Each CWE entry contains a detailed description of the weakness, including its name, description, potential consequences, examples of how it can be exploited, and

mitigation strategies. The weaknesses in the CWE repository are organised into a hierarchical structure, making it easier to navigate and understand the relationships between different types of weaknesses. Additionally, CWE is integrated with other cybersecurity standards and frameworks, such as CVE and the National Vulnerability Database (NVD) [43], enhancing its utility and interoperability.

The Mitre CWE repository is also a beneficial resource for mobile application developers, enabling them to proactively tackle potential security vulnerabilities in their source code. As shown in [17], this knowledge is crucial for early vulnerability detection. A list of common vulnerabilities discovered in Android code, can be found in Table 2.1 [42].

Table 2.1: Common Vulnerabilities in Android Code

CWE ID	**CWE Description**
CWE-79	Improper Neutralisation of Input During Web Page Generation ('Cross-site Scripting')
CWE-89	Improper Neutralisation of Special Elements used in an SQL Command ('SQL Injection')
CWE-200	Exposure of Sensitive Information to an Unauthorised Actor
CWE-295	Improper Certificate Validation
CWE-297	Improper Validation of Certificate with Host Mismatch
CWE-327	Use of a Broken or Risky Cryptographic Algorithm
CWE-330	Use of Insufficiently Random Values
CWE-599	Missing Validation of OpenSSL Certificate
CWE-649	Reliance on Obfuscation or Encryption of Security-Relevant Inputs without Integrity Checking
CWE-676	Use of Potentially Dangerous Function
CWE-926	Improper Export of Android Application Components
CWE-927	Use of Implicit Intent for Sensitive Communication
CWE-939	Improper Authorisation in Handler for Custom URL Scheme
CWE-250	Execution with Unnecessary Privileges
CWE-276	Incorrect Default Permissions
CWE-299	Improper Check for Certificate Revocation
CWE-312	Cleartext Storage of Sensitive Information
CWE-502	Deserialisation of Untrusted Data
CWE-532	Insertion of Sensitive Information into Log File
CWE-919	Weaknesses in Mobile Applications
CWE-921	Storage of Sensitive Data in a Mechanism without Access Control
CWE-925	Improper Verification of Intent by Broadcast Receiver
CWE-749	Exposed Dangerous Method or Function

2.4 Machine Learning Processes

In recent years, there has been a surge in the application of machine learning (ML) techniques for detecting vulnerabilities in software [5]. This surge underscores the necessity for a solid understanding of ML procedures to enhance our comprehension of studies focused on detecting source code vulnerabilities using ML. The lifecycle of ML encompasses several critical stages, each integral to the development of robust ML models.

The process begins with data extraction, where relevant data is gathered from various sources. This raw data often contains anomalies such as missing values, outliers, and redundancies, which need to be addressed during preprocessing. Preprocessing transforms the raw data into a format suitable for machine learning algorithms by cleaning and normalising it. This step ensures that the data is consistent and free of errors that could compromise the accuracy of the model. Following preprocessing, feature selection takes place. This stage involves identifying the most relevant features or variables that will contribute to the predictive power of the model. Feature selection is crucial as it helps simplify the model, reduce computation time, and avoid over-fitting by eliminating irrelevant or redundant features.

Once the features are selected, the data is used to train the model. During training, the model's parameters are adjusted to minimise a loss function, which quantifies the difference between the predicted and actual outcomes. This stage involves iterative optimisation techniques to find the best parameter settings that enhance the model's performance. After training, the model is evaluated using a distinct testing dataset. This evaluation is essential to gauge the model's performance and involves metrics such as accuracy, precision, recall, or F1 score. These metrics provide insights into how well the model generalises to new, unseen data. Once the model's performance meets the desired criteria, it is deployed in a real-world setting to make predictions on new data. However, the lifecycle does not end with deployment. The model must be continuously monitored and updated as necessary to maintain optimal performance and adapt to new data or changing environments [44].

Machine learning encompasses various types, each suited to different kinds of tasks. These include supervised learning, unsupervised learning, semi-supervised learning, reinforcement learning, and deep learning. In supervised learning, models are trained using labelled datasets to solve classification and regression problems. This involves algorithms such as Naive Bayes (NB), Logistic Regression (LR), Linear Regression, Gradient Boosting (GB), Extreme Gradient Boosting (XGBoost), Support Vector Machine

(SVM), Decision Tree (DT), Random Forest (RF), and k-Nearest Neighbours (kNN). Supervised learning algorithms learn from the training data by mapping input features to the corresponding output labels, allowing them to make accurate predictions on new data. Unsupervised learning, in contrast, deals with datasets that do not have labels. It uncovers hidden patterns and structures within the data through techniques like clustering, association, and dimensionality reduction. Common algorithms include K-means clustering, Principal Component Analysis (PCA), and auto-encoders. These methods are particularly useful for tasks such as market segmentation, anomaly detection, and data compression. Semi-supervised learning combines elements of both supervised and unsupervised learning, making it valuable when limited labelled data is available. This approach can improve learning accuracy by leveraging the abundance of unlabelled data alongside the labelled data.

Reinforcement learning involves training models based on feedback from their environment. Unlike supervised learning, it does not rely on labelled training data. Instead, models learn to make decisions by taking actions in an environment to maximise cumulative rewards. This trial-and-error approach is widely used in applications like robotics, games, and autonomous driving. Deep Learning (DL), a subset of ML, is characterised by its use of neural networks with many layers. DL models can automatically discover intricate patterns in large datasets through hierarchical feature learning. Some widely used DL algorithms include Convolutional Neural Network (CNN), Long Short Term Memory Network (LSTM), Recurrent Neural Network (RNN), Generative Adversarial Network (GAN), and Multilayer Perceptron (MLP). These models excel in tasks such as image and speech recognition, natural language processing, and generative tasks [45].

2.5 XAI for Interpreting AI Prediction Results

Traditional AI models often yield predictions that are not easily interpretable, functioning much like a black box. This lack of transparency poses significant challenges, especially when these models are deployed for tasks such as detecting vulnerabilities in software applications. Without a clear understanding of how predictions are made, app developers struggle to grasp the underlying reasons for the identified vulnerabilities. Consequently, they find it difficult to develop targeted countermeasures to address these issues effectively. This opaque nature of traditional AI models means that developers have to invest additional effort and resources to interpret the results, detracting from their primary focus on app development.

To address this issue, the implementation of Explainable AI (XAI), as explored in [46], emerges as a practical approach. XAI methods enhance the transparency of AI models by providing human-understandable explanations for their predictions. These explanations demystify the decision-making processes of AI models, enabling users to comprehend the logic and rationale behind specific predictions or decisions. By doing so, XAI facilitates a deeper understanding of the model's behaviour, which is crucial for identifying the root causes of code vulnerabilities.

For app developers, employing XAI can significantly streamline the vulnerability detection and resolution process. With clear explanations at hand, developers can quickly pinpoint the specific aspects of the code that are problematic, understand why these vulnerabilities were flagged by the AI model, and devise appropriate countermeasures with greater efficiency. This not only enhances the accuracy and reliability of vulnerability detection but also integrates seamlessly with the developers' workflow, minimising the additional effort required to interpret AI predictions.

Moreover, XAI contributes to building trust in AI systems. When developers and other stakeholders can see and understand how an AI model arrives at its conclusions, they are more likely to trust and rely on these systems. This trust is essential for the broader adoption of AI technologies in sensitive and critical applications, such as cybersecurity and software development.

In summary, the incorporation of Explainable AI methods addresses the interpretability issue inherent in traditional AI models. By offering clear and understandable explanations for AI predictions, XAI not only helps developers to efficiently tackle code vulnerabilities but also fosters greater trust and reliance on AI systems. The advancements in XAI, mark a significant step towards more transparent, accountable, and effective AI applications in software development and beyond.

2.6 Data Scarcity

Data scarcity in the development of AI models pertains to the inadequacy of data required for the effective training of machine learning algorithms. This deficiency includes a shortfall in the volume, quality, and diversity of datasets needed for robust model training [47]. Data scarcity poses significant challenges as it limits AI models' ability to generalise to new, unseen data, often leading to over-fitting where models become overly adapted to the training data [48, 49]. To address the issue of over-fitting, cross-validation

techniques such as 5-fold cross-validation, as applied in this work, can be utilised. These techniques help ensure the model performs consistently across different data subsets, confirming its ability to generalise to unseen data. Additionally, early stopping, as proposed in this work, can be implemented to prevent over-fitting by monitoring the model's performance on a validation set during training and stopping when performance begins to decline. Furthermore, ensemble methods, including bagging, boosting, and stacking, can be employed to improve generalisation and reduce variance in the model's predictions.

Constraints in data scarcity compromise the models' predictive precision and reduce their utility in real-world situations. Furthermore, data scarcity limits the complexity and capabilities of AI models, obstructing their ability to detect and represent complex patterns in the data.

Various approaches can be utilised to tackle data scarcity. Gathering real-world data from a range of sources can mitigate data scarcity by supplying more extensive and representative datasets for training models [50]. However, the acquiring real data may trigger privacy issues, underscoring the necessity to balance the efforts to counteract data scarcity with the need for ethical data practices. Therefore, while it is essential to address data scarcity to improve the performance of AI models, it should be done responsibly, with due consideration for privacy implications and ethical factors. Additionally, techniques for data augmentation can also be considered, which include the creation of synthetic data or the modification of existing samples, which can increase the size and diversity of datasets, thereby bolstering model resilience [50, 51].

2.7 Federated Learning and Differential Privacy for AI Models

Federated learning is a type of distributed machine learning where many local models are trained on different devices to build a global model. In this setup, clients connected to a server train their own local models using their data across several training cycles. During these cycles, the weights of the models are sent to the federated server. Here, they are averaged and updated, leading to the formation of a global model. This process uses the Federated Averaging (FedAvg) algorithm.

FedAvg, as described in Algorithm 1, is a commonly used method in federated learning that allows for the training of local models on multiple clients without needing to share the client's raw data with the server [52]. The ability to achieve model convergence across

a variety of client datasets, even in settings where the data is not independent and identically distributed, is crucial in federated learning [53]. Notably, federated learning only shares model weights with the federated server, not the original data, which effectively safeguards the privacy of the client data [54].

Algorithm 1: Federated Averaging (FedAvg) Algorithm

Input: N: Total number of clients
K: Number of communication rounds
w_0: Initial global model
α: Learning rate
Result: Updated global model w_K

for $k = 1$ **to** K **do**
  for $i = 1$ **to** N **do**
    Train a local model $w_{i,k}$ using client i's local data:
    $w_{i,k} = \text{LocalTraining}(w_k, \alpha)$
  end
  Aggregate local model updates:
  $w_{\text{agg}} = \frac{1}{N} \sum_{i=1}^{N} w_{i,k}$
  Update the global model:
  $w_{k+1} = w_{\text{agg}}$
end

Even with the use of federated learning for AI model training, there are ongoing concerns about the privacy implications of sending model updates to a central server. While federated learning greatly protects clients' private data from exposure to external parties, private information can still be revealed by analysing parameters uploaded to the central server or weights trained and shared in neural networks. To mitigate this, organisations can bolster privacy protection by integrating differential privacy mechanisms into their federated learning procedures [55, 56].

The process involves using differential privacy methods in the consolidation of model updates or parameters, thereby ensuring the privacy of individual clients' data throughout the federated learning process. By implementing this combined approach, the likelihood of private information leakage is reduced, thus strengthening the overall privacy stance of federated learning models. This enhancement not only protects sensitive data but also fosters increased trust among stakeholders regarding the privacy-preserving abilities of federated learning systems. As organisations traverse the changing terrain of data privacy laws and user expectations, the incorporation of differential privacy alongside federated learning signifies a forward-thinking move towards encouraging responsible and ethical

AI development practices.

Differential privacy is a system designed to protect individual data within a dataset, while still allowing valuable insights to be extracted. This method involves introducing measured noise into the training data, ensuring that even if an adversary were to access the results of data analysis, they would be unable to determine whether specific individuals contributed to the dataset [57]. This safeguard minimises the potential for privacy violations and unauthorised access to confidential information. However, the introduction of noise into the training procedure can affect the overall performance of AI models, requiring a delicate equilibrium between preserving privacy and maintaining model functionality [58].

The Tensorflow privacy [59], as outlined in algorithm 2 , is beneficial for ML problems when implementing differential privacy. The algorithm accepts the training data and privacy parameters (ϵ, δ) as input and produces differential private model parameters as output. Throughout each training epoch, the algorithm cycles through mini-batches of the training data. For each mini-batch, it calculates the gradients of the model's loss function relative to the parameters, trims the gradients to a predetermined norm to lessen the influence of outliers, introduces noise to the gradients to guarantee differential privacy, and modifies the model parameters using a gradient descent step. Ultimately, it yields the trained model parameters. This procedure ensures that the model acquires knowledge from the data while safeguarding the privacy of individual records.

Algorithm 2: Differential Privacy Algorithm using TensorFlow Privacy

Input: Training data $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_n, y_n)\}$, Privacy parameters ϵ, δ

Output: Differentially private model parameters

Initialize model parameters θ randomly;
for *each epoch* **do**
  for *each minibatch* $(\mathbf{x}_i, y_i)$ *in the training data* **do**
    Compute gradients: $\nabla_\theta \ell(\mathbf{x}_i, y_i, \theta)$;
    Clip gradients: $\hat{\nabla}_\theta = \text{Clip}(\nabla_\theta, \text{clip\_norm})$;
    Add noise to gradients: $\tilde{\nabla}_\theta = \hat{\nabla}_\theta + \mathcal{N}(0, \sigma^2 \mathbf{I})$;
    Update model parameters: $\theta \leftarrow \theta - \eta \tilde{\nabla}_\theta$;
  end
end
return θ

2.8 Blockchain-based Federated Learning

Traditional federated learning faces inherent challenges in coordinating participant activities, determining their incentives, and aggregating models. These issues arise primarily from its centralised approach, which relies on a trustworthy central authority for effective coordination. This method introduces various disadvantages, including susceptibility to attacks, credibility concerns, complexities in reward calculation, and difficulties in motivating participants for model training. One approach to address these challenges is Multi-party Computation (MPC), enabling multiple parties to collaboratively compute functions over their respective inputs while preserving data privacy. This approach fosters collaboration without the need to share raw data, thereby mitigating privacy concerns and reducing dependence on centralised control. Alternatively, homomorphic encryption allows computations on encrypted data without decryption, maintaining data privacy during essential operations such as model aggregation and reward computation. Secure version control techniques using platforms like GitHub repositories and subversion can also provide a reliable method for managing federated learning processes. However, challenges such as computational overhead, privacy limitations, and transparency concerns persist despite these alternatives. As a promising solution, blockchain technology has emerged as a potential remedy, offering secure and transparent mechanisms to address these complexities effectively.

A blockchain is essentially a distributed ledger build with a series of data blocks arranged in sequence, each block containing a set of verifiable transactions [60]. Except for the first block, each block in a blockchain includes the hash value of the previous block's header, forming a chain of linked blocks. Blockchains use unique consensus mechanisms like proof of work (PoW) or proof of stake (PoS), managed by a Peer-to-Peer (P2P) network of nodes. This structure makes it difficult to create but easy to verify each data block. The chain's structure, the complexity of block creation, and the decentralised consensus achieved in P2P networks make the data in a blockchain highly resistant to changes. The unchangeable nature and traceability of data in blockchains make them a robust technical solution for maintaining a reliable database in a decentralised and trustworthy way [61].

As a result, researchers from various disciplines have examined methods of federated learning based on blockchain from different perspectives [14, 62]. However, none of these studies have yet investigated the possible use of blockchain in conjunction with federated learning for detecting vulnerabilities in Android code using AI.

2.9 Chapter Summary

This chapter laid the background for the research by discussing the security risks and vulnerabilities associated with Android, vulnerabilities in source code, the workings of the machine learning process, and how XAI can be leveraged to rationalise machine learning predictions. It also discussed the issue of data scarcity and the strategies that can be employed to mitigate it, the principle of federated learning, differential privacy, and the integration of blockchain to facilitate a community-driven model. Consequently, readers can gain a comprehensive understanding of these areas, which are integral to this doctoral research.

Chapter 3

Literature Review

As outlined in Chapter 1, Android applications frequently undergo inadequate security assessments, underscoring the imperative to revise the development process to align with rigorous security standards, unlike iOS, which serves as Android's primary competitor. Hence, this chapter delves into the analysis of Android applications, and the detection of source code vulnerabilities, by critically reviewing selected technical research, focusing on answering to RQ1 and towards achieving RO1. It underscores the pros, cons, and practicality of the suggested techniques, as well as potential enhancements to these studies. The discussion encompasses both traditional and ML-based methods for vulnerability detection, with a greater emphasis on ML-based methods due to their prominence in recent research.

The review was carried out with the aim of finding answers to the following questions:

RQ1.1: What are the current techniques for analysing source code and applications?

Numerous research studies have explored different methods of source code analysis, including the reverse engineering of applications. In addition, analysers based on byte-code are commonly used, as Android apps can be readily reverse-engineered back to source code. Static analysis techniques have been widely employed, along with dynamic and hybrid analysis techniques for source code analysis. These methods are further discussed in section 3.3.

RQ1.2: What are the current methods for detecting vulnerabilities in Android source code, and how can they be utilised to mitigate these vulnerabilities?

In the detection of vulnerabilities in Android source code, both ML and traditional methods have been utilised in numerous studies. The use of ML methods has gained popularity in recent years within the research community, leading to their application in many studies. On the other hand, a smaller number of studies have employed traditional methods that do not involve ML. However, merely identifying vulnerabilities is not enough to enhance the security of Android source code. It is also necessary to explore how to prevent these security issues by incorporating detection techniques into software development environments. These methods of detection and prevention are further discussed in section 3.4.

RQ1.3: How can XAI be utilised to interpret the results of AI-based predictions?

Receiving a prediction as a black-box does not provide much insight into the underlying reasons for the prediction or how to devise potential mitigation strategies in the event of vulnerable source code. Explainable AI can serve as a supportive mechanism for this purpose. This topic is further discussed in section 3.5.

RQ1.4: What are the tools and repositories available for identifying vulnerabilities in Android applications?

Investigating the tools, repositories, and datasets that can be employed for source code analysis and vulnerability detection is crucial. Recognising their features and how they are used can aid in carrying out new research studies. These aspects are discussed further in section 3.6 and section 3.7.

3.1 Conducting the Literature Review

The initial phase of the research involves a comprehensive review of relevant literature, including the identification of AI/ML models that can bolster application security. This step aligns with RO1, as elaborated in chapter 1. The Preferred Reporting Items for Systematic Reviews and Meta-Analysis (PRISMA) model [63] was employed to report and analyse the research studies in this field. The search strategy was defined based on the objectives of this study, aiming to identify studies that could answer the defined literature

search questions. The usage of databases and the criteria for inclusion and exclusion were also established. Subsequently, the selection of studies, data extraction, and synthesis were carried out to identify studies aiming to answer the formulated questions.

3.1.1 Search Strategy

The review process began with an analysis of existing literature on the detection of malicious code and vulnerabilities in Android, aiming to pinpoint the research gap. After identifying this gap, a search string was utilised to extract and pinpoint technical studies pertinent to the focus of the review.

The search strategy entails defining the most pertinent bibliographic sources and search terms. This review employed numerous leading research repositories, such as the ACM Digital Library, IEEEXplore Digital Library, Science Direct, Web of Science, and Springer Link, as the main sources for identifying studies. The search query *((("vulnerability detection") OR ("source code vulnerability") OR ("vulnerable code") OR ("code vulnerability detection") OR ("vulnerability analysis") OR ("static analysis") OR ("dynamic analysis") OR ("hybrid analysis") OR ("vulnerability dataset")) AND ("android") AND (("machine learning") OR ("deep learning") OR ("formal methods") OR ("heuristic methods")))* was used to navigate through these research repositories.

Several years after Android's initial release in 2008, the growing popularity of Android applications led to increased discussions about security concerns [64]. Numerous studies proposed methods to detect and prevent vulnerabilities, thereby enhancing software security using both ML and non-ML-based methods. Over the past five years, there has been a surge in the application of various techniques to bolster application security [65]. Trends related to detecting vulnerabilities in mobile applications using ML techniques have been on the rise since 2016 [5]. Consequently, many researchers are engaged in discovering innovative ML-based methods to improve software security. Given these factors, technical studies from 2016 to 2024 were reviewed. The distribution of search results across primary sources for each search term is presented in Table 3.1.

In addition to the primary repositories, Google Scholar was utilised as an alternative source to identify research studies published in reputable venues, as it can help to discover studies not published in the main repositories. The search query used was *Android source code vulnerability detection*, and the publication years were set from 2016 to 2024. Although the search yielded approximately 18,250 records, only the top 150-160 results (sorted by relevance) for each year were taken into account, leading to 1,400 studies.

Table 3.1: Distribution of Search Results from Primary Sources from 2016 to 2024

Search Term	**ACM**	**IEEEXplore**	**Science Direct**	**Web of Science**	**Springer**	**Total Count**
vulnerability detection	431	381	272	568	512	**2,164**
source code vulnerability	14	15	19	14	39	**101**
code vulnerability detection	10	12	9	14	22	**67**
vulnerability analysis	388	340	2,521	1,651	752	**5,652**
vulnerable code	284	101	91	105	203	**784**
static analysis	5,599	2,219	2,615	3,397	4,751	**18,581**
dynamic analysis	2,661	1,955	3,071	2,212	2,791	**12,690**
hybrid analysis	179	91	162	141	263	**836**
vulnerability dataset	24	14	27	11	45	**121**
android	12,551	8,951	5,361	9,955	12,691	**49,779**
machine learning	55,891	90,451	148,321	291,024	89,481	**675,168**
deep learning	24,526	72,812	67,122	129,123	49,281	**342,864**
formal methods	3,499	1,389	1,721	3,921	5,869	**16,399**
heuristic methods	771	1,279	3,391	1,741	3,488	**10,670**
Complete Search String	**724**	**159**	**611**	**261**	**981**	**2,736**

3.1.2 Study Selection, Data Extraction, and Synthesis

At the outset, a search was conducted in premier research databases and Google Scholar, yielding 2,736 and 1,400 research papers, respectively. Out of these 4,136 papers, 3,291 were discarded due to duplication, and 129 were excluded as they were not publicly accessible. This left 716 studies after the preliminary screening. It is worth noting that research repository search engines often yield irrelevant results [66]. To refine the list of pertinent studies, each paper was manually reviewed by examining its title and abstract to ensure alignment with the review's focus. This process resulted in 133 eligible studies. However, four were further excluded due to data analysis and experimentation issues in the given context, leaving 129 studies. A snowballing process [67] was employed, which involved reviewing all references in the retrieved papers and all papers citing the retrieved ones. This process added two more relevant papers. After all these steps, a total of 131 articles were selected to review. The findings were then cross-verified through a peer-verification process. A summary of the paper selection methodology for this systematic review is presented in Figure 3.1.

3.1.3 Threats to Validity of the Review

Although this systematic review adhered to a well-established methodology [63], there is no guarantee that all relevant studies were included due to certain limitations in the review process. Therefore, this section addresses potential threats to validity and the measures taken to minimise them. These are categorised under construct, internal, external, and conclusion validity.

Construct Validity

Threats to construct validity may arise from the search term-based queries performed on various repositories. Some relevant papers might not have been reviewed because they were unavailable in the research repositories, including the ACM Digital Library, IEEEXplore Digital Library, Science Direct, Web of Science, and Springer Link. To mitigate this, Google Scholar was also used as an additional source to capture potentially missing studies. Despite these efforts, some relevant publications may still be absent from the collected studies. Another aspect of construct validity concerns the possibility of errors in applying the inclusion or exclusion criteria when filtering studies. To avoid these errors, the list of publications was analysed by cross-checking the primary studies.

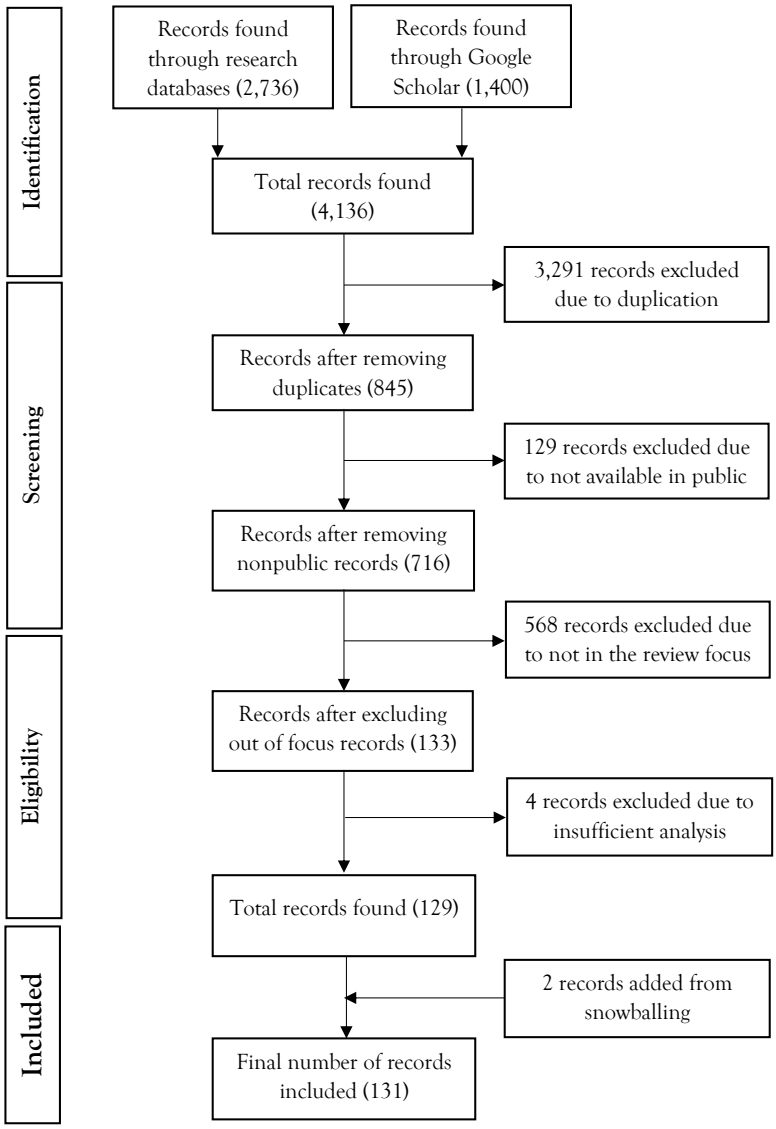


Figure 3.1: PRISMA method: collection of papers for the review

Internal Validity

Internal validity pertains to the accuracy of data extraction and analysis, focusing on the robustness of the proposed review process. Due to the significant workload involved, data extraction and analysis were cross-checked, and consensus was reached among all authors on the comparison results. However, some errors in data extraction and analysis are still possible. Involving the original authors in the verification process could further minimise these mistakes.

External Validity

External validity concerns the generalisability of the results obtained from the primary studies. The analysis in this review focuses on research publications collected from 2016 to 2024, covering Android code vulnerability detection methods up to the present. During this period, the use of ML techniques for vulnerability detection has grown significantly due to recent advances in software security and artificial intelligence. Trends may vary for different time periods, so this review may not include some comprehensive studies conducted before this time frame.

Conclusion Validity

Efforts were made to minimise bias and other factors affecting this review study during the paper search process. Only research papers written in English were considered, which may have resulted in overlooking significant works written in other languages, such as Chinese, German, and Spanish. Additionally, there is a potential threat of bias from the consideration of studies with individual biases, which could introduce flaws in this study. Positive results are more likely to be reported than negative ones [68]. Nevertheless, many papers reporting adverse effects were included, as a peer-verified systematic review process was followed. A cross-checking mechanism was also employed to ensure a thorough examination and maintain the focus of the review when evaluating the papers.

3.2 Related Literature Reviews

Existing literature reviews [69, 70, 71, 72, 73, 74, 75, 76] have explored a range of security-focused research, encompassing methods for identifying vulnerabilities in Android applications and strategies for mitigating them. Given the swift emphasis on software security, the majority of these investigations and trials were carried out post-2015.

The study on Android's security was examined in [70]. This review encompassed various Android threats, including information leakage, privilege escalation, app repackaging, denial of service attacks, and collusion. It also evaluated methods such as Crowdroid [77], Kirin [78], AndroSimilar [79], RiskRanker [80], RiskMon [81], FireDroid [82], Aurasium [83], DroidScope [84], RecDroid [85], and DroidRanger [86]. These studies were scrutinised with goals such as evaluation, source code analysis method (i.e., static or dynamic analysis), and detection mechanisms. Nonetheless, this review has its limitations,

such as adopting an informal review approach and not providing exhaustive details on mechanisms for detecting and preventing vulnerabilities.

Recognising duplicate code across various parts of a program is of paramount importance. When bugs are identified in a program, it is also essential to pinpoint areas of code repetition, as all these locations need to be revised. The research in [73] conducted a review of numerous studies on code clone detection. This study systematically examined 54 studies, categorised under six headings: textual approaches, lexical approaches, tree-based approaches, metric-based approaches, semantic approaches, and hybrid approaches. This review also identified twenty-six clone detection tools, and it was found that many of these tools and models are applicable to Java/ C++ codes. However, as this review encompasses papers from 2013 to 2018, it would be beneficial to review the most recent code clone detection methods proposed post-2018.

The authors of [76] conducted an analysis of 55 studies from 2015 to 2021 that focused on software vulnerability detection. The chosen papers were categorised based on various evaluation criteria for vulnerability detection, including neural networks, machine learning, static and dynamic analysis, code cloning, classification models, and frameworks. The analysis revealed that a significant number of researchers employed machine learning techniques for software vulnerability detection, given its ability to easily analyse large volumes of data. While some of the studies reviewed have overlapped with vulnerability detection in C and Java source code, it is recommended that a separate review be carried out specifically for detecting vulnerabilities in Android source code.

The research in [74] conducted a review of Android security evaluations, which included the examination of trends and patterns across various analysis methods, techniques, code representation tools, and relevant frameworks by analysing approximately 200 studies from 2013 to 2020. The study also delved into issues such as privacy leaks, cryptographic problems, app cloning, misuse of permissions, code verification, malware detection, test case generation, and energy consumption. Under the umbrella of static analysis techniques, the study discussed sensitivity analysis, data structures, and code representations. Inspections at the kernel level, application level, and emulator level were also considered under taint analysis and anomaly-based approaches in dynamic analysis techniques. The review underscored that numerous research studies were conducted concerning Android vulnerabilities and leaks. Furthermore, this study systematically reviewed several Android assessment techniques and identified call graphs, control flow

graphs, and inter-procedural control flow graphs as the utilised data structures. However, the studies related to vulnerability prevention were not discussed. Additionally, it is feasible to review studies on non-ML-based methods for detecting and preventing vulnerabilities, as this review only considered ML methods.

The research in [72] reviewed studies related to automated testing mechanisms for Android apps from 2010 to 2016. This paper delved into three types of functional testing: black-box, white-box, and grey-box, by analysing Android-related studies. These studies included test-related objectives, targets, levels, techniques, and their validation depths. The test objectives considered were bugs, defects, compatibility, energy, performance, security, and concurrency. In terms of test targets, it took into account inter-component communication, inter-application components, graphical user interface, and events. System, integration, and regression were considered under test levels, while testing types, testing environments, and testing methods were listed as used test techniques. Additionally, the execution of tests using emulators and real devices was discussed. It scrutinised testing methods including mutation, concolic, A/B, fuzzing, random, search-based, and model-based. This review took into account frequently used essential tools such as AndroidRipper [87], Monkey [88], Silkuli [89], Robotium [90], EMMA [91], and Roboelectric [92]. However, it is worth noting that while it has comprehensively reviewed app testing up until 2016, recent studies have not been considered.

The review in [71] analysed 124 research studies from 2011 to 2015 with the aim of discerning static analysis mechanisms for Android applications. It was found that static analysis was employed in numerous research studies pertaining to privacy and security, with taint analysis being the technique most frequently applied in these studies. As per the review, Soot - a framework for analysing, instrumenting, optimising, transforming, and visualising Java and Android applications [93], and Jimple - an intermediate representation that simplifies the analysis and transformation of Java bytecode [94] were the tools and formats most commonly used, and a handful of studies took path-sensitivity into account. Upon analysis, this review determined that leaks and vulnerabilities were the primary concerns addressed by the other research studies. Furthermore, this review discovered issues related to permission misuse, energy consumption, clone detection, test case generation, code verification, and cryptographic implementation. However, recent techniques, including those related to ML, were not reviewed as this review was focused on research from 2011 to 2015.

The study in [75] conducted a review of DL-based defences against Android malware by addressing three main research questions: 1) the aspects of Android malware defences that are applied when using DL, 2) the approaches developed for malware defences, and 3) the emerging and potential research trends for DL-based Android malware defences. The review took into account technical studies from 2014 to November 2021. It was found that while many of the studies reviewed primarily focused on DL-based Android malware detection, some defence approaches were based on non-DL methods. It was also noted that static program analysis is commonly used to gather features, and semantic features frequently appear. Furthermore, it was concluded that most of the approaches were carried out as a supervised classification task. This review recognised that numerous studies were conducted to detect malware, and more detailed analyses of malicious apps are gaining increased attention. However, it did not thoroughly review how other types, such as malicious code detection and code vulnerability detection, can be executed.

The systematic review carried out in [16] explored ML-based and DL-based methods for detecting Android malware, along with a comparative analysis of these methods and their accuracies. This review scrutinised numerous studies from 2017 to 2021 and determined that static, dynamic, and hybrid analyses could be employed with ML/ DL models for malware detection. Moreover, it was found that static analysis was the most frequently used technique in the studies reviewed. It was discovered that RF, SVM, NB, kNN, LSTM, and AdaBoost (AB) were the ML and DL models most commonly used in this context. In addition to the malware detection method, this review briefly touched upon the identification of Android software vulnerabilities. It examined the methods and techniques used to identify source code vulnerabilities. The study identified that hybrid analysis techniques were predominantly used to detect Android source code vulnerabilities. While the main focus of this review was Android malware detection using ML/ DL, it is still crucial to conduct a more comprehensive review of code vulnerability detection methods.

Research related to the Android security framework, evaluations of its security mechanisms, and strategies for mitigation were examined in [69]. The review covered security mechanisms including user interfaces, file access, memory management, type safety, mobile carriers, application permissions, component encapsulation, and application signing. It also reviewed security analysis studies pertaining to cornerstone layers of the Android framework, application-level permissions, application installation, mobile web browsers, SQL injections, connectivity and communication, hardware, software updates, malware

in the Linux environment, and Java-related malware. In terms of mechanisms, the review considered studies on anti-malware tools, firewalls, intrusion detection and prevention methods, access controls, permission management applications, encryption methods, and spam filters. Although this review discussed studies conducted on Android security using an informal and non-systematic approach, it did not take into account security issues such as API vulnerabilities, concurrency bugs, and the most recent OS-related bugs due to the time period considered in the review.

While these existing reviews offer comprehensive insights into the related studies, reviews like [71, 72] did not encompass the latest research conducted in this field. Reviews such as [69, 73] did not provide an exhaustive review of the studies on Android-specific vulnerability detection that employed various experiments in source code analysis. Table 3.2 provides a summary and comparison of the related reviews.

Therefore, it is required to conduct a comprehensive review of recent technical studies related to Android source code vulnerability detection and prevention mechanisms, to identify the exact answers to the questions formulated at the beginning of this chapter.

3.3 Application Analysis

The initial phase in identifying vulnerabilities should involve an analysis of the applications or source code [95]. Two primary methods of analysis exist: one involves examining the reverse-engineered source code of APKs, and the other involves concurrent analysis of the source code during its creation. The majority of research has focused on the reverse-engineering method. However, the latter method offers more benefits to developers as it allows for early detection of code vulnerabilities during development. It is crucial to review studies on both methods due to the significant overlap in their methodologies.

The analysis of applications or source code is the initial step in identifying vulnerabilities [95]. There are two main strategies for this analysis: one involves analysing the reverse-engineered source code of Android APKs, and the other involves analysing the source code in real-time as it is being written. While most research has been focused on the reverse-engineering method, the real-time analysis method offers more benefits to developers by allowing them to spot code vulnerabilities early in the development cycle. It is important to examine studies on both methods due to the considerable overlap in their methodologies.

Table 3.2: Summary of Related Reviews

Paper	Focus of the Review	Period	Review Approach	Number of Reviewed Studies	Android Specific	Static Analysis	Dynamic Analysis	Hybrid Analysis	ML-based code vulnerability detection	Non ML-based code vulnerability detection	Code vulnerability prevention	Supportive tools and repositories
Ahmed et al. [70]	Security in Android platform, associated threats, and malicious application growth	2010-2017	Informal	14	✓	✓	✓	✓	x	x	x	x
Ain et al. [73]	Code clone detection for vulnerabilities	2013-2018	Systematic search (Budgen [66] and Kitchenham [68] guidelines)	54	x	✓	✓	✓	x	✓	x	x
Eberendu et al. [76]	Various methods for detecting software vulnerabilities	2015-2021	Systematic search - (PRISMA model [63])	55	x	✓	✓	✓	✓	x	x	x
Grag et al. [74]	Android security assements and application analysis methods	2013-2020	Systematic search (Kitchenham guidelines [68])	200	✓	✓	✓	✓	✓	x	x	✓
Kong et al. [72]	Automated testing mechansims for Android	2010-2016	Systematic search (Kitchenham guidelines [68])	103	✓	✓	✓	x	x	x	x	✓
Li et al. [71]	Static analysis for Android apps	2011-2015	Systematic search (Kitchenham guidelines [68])	124	✓	✓	x	x	x	✓	x	x
Liu et al. [75]	Malware and malicious code detection with DL	2014-2021	Systematic search (Kitchenham guidelines [68])	132	✓	✓	✓	✓	✓	x	x	x
Senanayake et al. [16]	Malware detection and malicious code detection with ML	2017-2021	Systematic search (PRISMA model [63])	106	✓	✓	✓	✓	✓	x	x	x
Shabtai et al. [69]	Android security framework and security assessment	2007-2009	Informal	42	✓	x	x	x	x	✓	x	x

The first step in analysing the source code of an application involves feature extraction. This can be achieved through three techniques: static, dynamic, and hybrid analysis [96, 97, 98]. Static analysis can be applied to reverse-engineered APKs, the application's source code, or byte code. However, static analysis alone cannot uncover all bugs and failures as it does not account for vulnerabilities that may arise during the app's execution. Dynamic analysis generates features by running applications and monitoring their behaviour with specific input parameters. However, this method can potentially crash the runtime environment due to severe vulnerabilities, and some vulnerabilities may go undetected [99]. The hybrid analysis technique incorporates elements of both static and dynamic analysis techniques, allowing it to analyse both the source code and the runtime behaviour of the application [100].

3.3.1 Static Analysis

Native Android applications can be developed using either Java or Kotlin, with Java being the more commonly used language. Other frameworks like React Native and Xamarin are also viable for developing Android mobile applications [101]. These applications typically include Extensible Markup Language (XML) files such as the Android Manifest, UI layouts, and other resources. Therefore, it is necessary to detect issues in both the source code and XML files. Static analysis can examine both types of files without needing to execute them.

The authors in [74] suggested five key areas for static analysis: analysis methods, sensitivity analysis, code representation, data structures, and levels of inspection. The analysis methods include symbolic execution, taint analysis, program slicing, abstract interpretation, type checking, and code instrumentation. Sensitivity analysis takes into account objects, contexts, fields, paths, and flows. Code representation utilises Smali [102], Dex-Assembler [103], Jimple [94], Wala-IR [104], and Java Byte code/ Java class. Data structures include the Call Graph, Control Flow Graph, and Inter-Procedural Control Flow Graph. The levels of inspection encompass kernels, applications, and emulators.

There are two methods of static analysis: manifest analysis and code analysis, which differ in their approach to feature extraction. Some research uses either manifest analysis or code analysis, while others use both [105]. Manifest analysis, a commonly used method of static analysis, can pull out elements like package names, permissions, activities, services, intents, and providers from the AndroidManifest.xml file. This file lists all the permissions an application uses, sorted into categories like dangerous, signature,

and normal. SigPID identified twenty-two permissions as significant, using a three-tier data purring method [106]. These tiers included support-based permission ranking, permission mining with association rules, and permission ranking with a negative rate. The second static analysis method, code analysis, focuses on the source code files. It can extract features like API calls, information flow, taint tracking, native code, clear-text analysis, and opcodes. The MaMaDroid method [107] serves as an example of API calls analysis. It abstracts apps' API call executions to form regular classes or packages using static code analysis techniques, then determines the call graph using the Markov chain.

3.3.2 Dynamic Analysis

Dynamic analysis, the second technique, involves examining the application by running it in a controlled sandbox environment. This method necessitates a finished product, such as an APK. As a result, it is commonly employed to identify vulnerabilities and malware in fully developed applications.

Dynamic analysis in [74] identified five techniques for feature extraction: network traffic analysis, code instrumentation, system call analysis, system resources analysis, and user interaction analysis. These methods were used to extract features related to network, process, usage, and component interactions. Network-related features included uniform resource locators, internet protocols, network protocols, network certificates, and network traffic. Process-related features considered non-encrypted data, Java classes, intents, and system calls. Usage-related features took into account aspects like processor, memory, battery, network, and process reports. User interaction analysis features considered elements like buttons, icons, actions, and events.

The authors in [108] employed dynamic analysis methods to detect vulnerabilities in Android. Their approach consisted of three components: collection of network traces, extraction of network features, and detection of network features. The collection module periodically recorded and monitored the network activities of active apps. The extraction module pulled out network features utilised in applications, such as features based on origin-destination, domain name system, transmission control protocol, and hypertext transfer protocol, and carried out the process of detecting vulnerabilities.

3.3.3 Hybrid Analysis

Hybrid analysis leverages both static and dynamic features to examine a given application. The research in [109] utilised static features such as permissions and intents, and

dynamic features like IPs, emails, and URLs, to gather diverse information about applications. The initial step involved using APKTool [110] to decompile the APK. Following data extraction, disassembled dex files were used to construct the feature vector for subsequent analysis. The APK files were run in an emulator to observe the behaviours of the dynamic features.

The model introduced in [111] employs a hybrid analysis to detect security vulnerabilities in Android. It uses static analysis to examine metadata and data flow, and dynamic analysis to inspect API hooks and executable scripts. The static analysis technique was capable of identifying eight categories of vulnerabilities: unrestricted component, insecure JavaScript in WebView, sensitive data processed in plain text, privacy leak by log, dynamically loading a file, insecure password, intent exposure, and SQL injection. The dynamic analysis technique could identify the category of unverified inputs vulnerability. However, it might fail if the app implements certain security measures like signature verification, leading to occasional false positives. Despite this, the overall analysis can be completed within an average of 93 seconds with an accuracy rate of approximately 95%. SSL/TLS issues are also crucial to detect and can be analysed using hybrid analysis. The DCDroid framework in [112] utilised hybrid analysis techniques for this purpose, finding that 360 out of 2,213 applications had security issues related to SSL/ TLS certificates.

3.4 Code Vulnerability Detection

Mobile applications can potentially exploit security mechanisms due to vulnerabilities in the source code [5]. While it is impossible to develop applications that are completely free of defects or vulnerabilities, efforts can be made to minimise them, and detecting vulnerabilities in the source code is a crucial step towards this goal. A variety of methods, including machine learning, deep learning, heuristic-based methods, and formal methods, can be employed to identify these vulnerabilities, utilising static analysis, dynamic analysis, and hybrid analysis techniques.

3.4.1 Using Machine Learning Methods

ML and DL techniques, including NB, LR, DT, RF, GB, LSTM, RNN, and MLP, have been utilised in studies for detecting vulnerabilities. To effectively train these ML or DL models, it is necessary to identify features in the Android application using an appropriate analysis method, which could be static, dynamic, or hybrid.

Machine Learning with Static Analysis

ML methods can be used in conjunction with static analysis techniques for detecting code vulnerabilities, provided the source code is transformed into a generalised form. The Abstract Syntax Tree (AST) is a commonly used method for this generalisation [113]. The frequency of false positives in vulnerability detection is influenced by the precision of the AST formulation and its generalisation mechanism, as well as the quality of the features, the chosen dataset, and the algorithms used for training. Research such as [5] has demonstrated the feasibility of using ML and DL methods on a generalised source code structure like AST for detecting vulnerabilities in Android code. Thus, enhancing feature generation methods like AST construction is identified as a research gap in this field for the application of ML techniques.

Several studies have employed static analysis techniques in conjunction with ML methods to detect malicious code and vulnerabilities. For instance, the WaffleDetector [114] uses a static analysis approach to identify malicious code and vulnerabilities in Android applications by examining sensitive permissions, program features, and API calls, with further analysis conducted using the Extreme Learning Machine (ELM). In another study [115], a framework named Vulvet was proposed for detecting and patching vulnerabilities. This framework employs static analysis techniques to identify vulnerabilities in Android applications and introduces a multi-tier, multi-pronged analysis technique. It also proposes an automated process for generating patches for vulnerabilities. To avoid false positives, the framework suggests augmented control-flow analysis and Android-specific component validation approaches.

The Vulvet framework [115] leverages features in the Soot framework, such as data-flow analysis, call-graph analysis, intermediate code scanning, taint analysis, parameter analysis, API analysis, and return value analysis. It also employs various techniques including vulnerability resolution, control-flow instrumentation, methods/parameters reconstruction, secure method call augmentation, manifest modification, and code elimination. This model has demonstrated its effectiveness, detecting vulnerabilities with 95.23% precision and a 0.975 F-Measure across 3,700 apps from benchmark datasets and other Android marketplaces. It was also found that 10.46% of the evaluated apps were vulnerable to various exploits. Despite its comprehensive nature, the model does have some limitations. For instance, it does not analyse and patch vulnerabilities in native code, does not support Java reflection and dynamic code loading, and marks all files read from external storage as malicious. These limitations present opportunities for further improvement.

The importance of data flow analysis in detecting malicious code and applications is highlighted in a study [116], which proposed a mining method for topic-specific data flow signatures to characterise malicious Android apps. The study found that these topic-specific data flow signatures were more effective than overall data flow signatures in characterising malicious and vulnerable apps.

Data flow patterns and descriptions were collected from 3,691 benign and 1,612 malicious apps for analysis. After extracting the features, a topic model was constructed using adaptive Latent Dirichlet Allocation (LDA) in conjunction with Genetic Algorithms (GA). GA was used to determine the optimal number of topics. Subsequently, a topic-specific data flow signature was generated by calculating the information gain ratio for each piece of data flow information. This information gain ratio was then used to characterise the apps. While this study considered a large number of apps, it did not take into account their representativeness, which could potentially reduce the accuracy of the process. This limitation could be addressed in future research by analysing a more representative set of apps and ensuring adequate sample sizes for each topic.

The source code, which can be extracted from the APK file or Portable Executable (PE) file, is subjected to static analysis. In [117], an automated method was proposed that classifies codes into malicious and secure categories using the PE structure. This method employed static analysis with RF, GB, DT, and CNN models, achieving a detection accuracy of 98.77%.

In another study [118], a model was developed to predict software code vulnerabilities prior to the application's release. The code was represented using an AST for analysis, and ML models were applied. The model was trained using Python, C, and C++ source codes from well-known datasets such as NIST [43], SAMATE [119], SATE IV Juliet Test Suites [120], and Draper VDISC [6]. However, a significant limitation of this approach was its inability to pinpoint the exact location of the vulnerable code segment.

The mechanism developed in [121] ML methods to categorise functions in the C language as either vulnerable or non-vulnerable. The process begins with the preparation of the AST, followed by data preprocessing, feature extraction, feature selection, and classification tasks using ML algorithms. This study utilised NVD to gather C language code blocks and their known vulnerabilities.

Another automated system for detecting vulnerabilities was proposed in [6], which uses C and C++ source codes. This system employs ML with deep feature representation

learning and compares the results with Bag of Words, using RF, RNN, and CNN. It uses existing datasets and the Drapper dataset [6], compiled using GitHub [122] and Drebin [123] repositories, which contain open-source functions and carefully selected labels.

While these studies ([117, 118, 6, 121]) considered Python, C, or C++ codes, the potential for applying these proposed approaches to detect code vulnerabilities in Android source code written in Java warrants further methodical investigation.

Machine Learning with Dynamic Analysis

Dynamic analysis techniques can also be utilised to train ML models to identify vulnerabilities during the execution of an application. The study [124] presented a dynamic analysis approach that employed ML models such as NB, K-Star, RF, DT and Simple Logistic to detect vulnerabilities and malicious applications. Features were extracted while the APKs were being run in an emulator. The Simple Logistic model demonstrated high performance with a precision of 0.997 and a recall of 0.996. However, some applications experienced crashes when executed in the emulator due to their dynamic behavior. The dataset used in the study requires further refinement to enhance accuracy, as there are shared permissions between malicious and benign applications that could potentially lead to incorrect classification.

The study in [125] utilised a dynamic analysis technique and discussed a mechanism for detecting code vulnerabilities by applying DL. It compared CNN, LSTM, and CNN-LSTM, finding that CNN-LSTM achieved a detection accuracy of 83.6%. It was also identified that Deep Neural Networks (DNN) can predict vulnerable source code. To classify the vulnerable classes with high precision, recall, and accuracy, the model proposed in [126] was used. This model was evaluated using Android apps written in Java. N-gram analysis and statistical feature selection were performed in this model to construct the feature vector.

Another study in [127] discussed a ML-based method for extracting vulnerability detection rules with dynamic analysis. The J48 ML algorithm performed with 96% accuracy, compared to thirty-two other supervised ML algorithms considered in this study. A context-aware intrusion detection system was proposed in 6th Sense [128], which used NB, Markov chain, and Logistic Model Tree (LMT) to detect vulnerabilities. This study observed changes in sensor-related data in the mobile device by integrating dynamic analysis methods. The model still requires some fine-tuning to the followed dynamic analysis approach to broaden vulnerability detection.

The dynamic analysis-based method introduced in [129] uses ML to detect anomalies in system calls by considering their type, sequence, and frequency. This method can identify Android security vulnerabilities by distinguishing between benign and malicious apps. Additionally, this work has created time-series datasets of system calls used in both vulnerable and regular applications.

The Zygote process of Android, which is responsible for creating new processes, was used in conjunction with the Android Debug Bridge (ADB) to trace every new activity and its associated processes. Common vulnerabilities in the selected application dataset included Dynamic Register Broadcast Receiver, Electronic Code Book (ECB) block cipher, fragment injection, weak permissions, and privilege escalation. Following this, a consolidated dataset was created by transforming unstructured time-series data. This dataset was then used to compute precision, recall, and F1-Score using kNN, LSTM, and the GA-LSTM. All three ML algorithms performed well, achieving over 85% F1-Score, with Genetic Algorithm LSTM slightly outperforming the others. Currently, this model can detect only nine types of vulnerabilities. Therefore, future studies should consider and verify more vulnerabilities while maintaining a similar accuracy to enhance the model.

Machine Learning with Hybrid Analysis

The application of hybrid analysis with ML methods is quite common as it enhances the detection approach by utilising both static and dynamic features. The research conducted in [130] introduced an ML-based mechanism for detecting vulnerabilities, employing hybrid analysis techniques and examining Android Intent mechanisms and their composition. Additionally, security detection related to Android Intents was explored by applying various ML algorithms such as DT, ID3, C4.5, NB, and AB. The model was tested on 150 applications with Android Intent mechanism security vulnerabilities and another 150 applications without them for training and testing. The proposed model achieved an average accuracy of 77%. However, limitations such as a small sample size and low performance were identified as areas that could benefit from further improvement.

The research in [131] introduced a parallel-classifier scheme for detecting vulnerabilities in Android. This study explored the potential of using unique parallel classifiers to identify zero-day malware and elusive vulnerabilities in Android, achieving an accuracy of 98.27%. It also highlighted some challenges in static and dynamic analysis approaches,

such as inefficiency, code obfuscation, and issues with the similarity score of signature-based detection. This model extracts static features like permissions, API calls, version, services, used libraries, broadcast receivers, and dynamic features like system calls, network calls of the mobile applications. It proposed an optimal combination of efficient ML algorithms such as SVM, Pruning Rule-Based Classification Tree (PART), MLP, and Ripple Down Rule Learner (RIDOR). While using parallel classifiers, this method also aimed to improve the precision and recall when detecting malware or vulnerabilities. Based on the initial results of the research, it was found that the MLP outperformed the other classifiers with a detection rate of 96.11

The subsequent part of the study used a composite model where the results from the initial part were executed in parallel to assess the efficiency of the cumulative approach. Ensemble techniques such as Average probabilities, Product of probabilities, Maximum probabilities, Majority vote, were considered. According to the final results, MaxProb emerged as the best parallel classifier. The study suggests that incorporating more parallel classifiers could enhance the model's accuracy, especially when using deep learning techniques.

Studies such as [132] have explored the potential of using ML algorithms in conjunction with both static and dynamic analysis to examine the source code in a hybrid manner. The primary objective of this approach was to distinguish between malware and benign applications by assessing their vulnerabilities. The process began with the extraction of data from APK files using the Androguard tool [133], which was then converted into a JavaScript Object Notation (JSON) file for static analysis. Datasets from Kaggle [134] and MalGenome [135] were utilised to train the ML models such as LR, SVM, and kNN. Subsequently, another JSON file was prepared to identify the code vulnerabilities. Finally, the APKs were dynamically analysed by executing them to uncover the vulnerabilities.

The model presented in [136] employed a hybrid analysis mechanism to detect malware and vulnerabilities using ML models. This model achieved an accuracy of 80% with the static analysis approach and 60% accuracy with the dynamic analysis approach. The findings suggest that integrating both these methods could enhance detection accuracy. However, a systematic approach would be beneficial to validate the increased accuracy when using hybrid analysis.

Another model, proposed in [137], utilised a hybrid analysis mechanism to identify vulnerabilities and malware. This model introduced a Tree-Augmented Naive Bayesian

Network (TAN) based mechanism that used features such as permissions, system and API calls. The output relationships were modeled as a TAN. It used datasets such as AZ [138], Drebin [123], Android Malware Dataset (AMD) [139], and GitHub [122]. The model demonstrated good performance with an accuracy of 97%. The primary limitation of the study was the consideration of only two features. Expanding the model with more features and training the dataset could lead to more reliable results. The feasibility of integrating into a single model rather than training separately could also be explored.

While numerous ML and DL based methods have been proposed, many of them do not possess the capability to detect code vulnerabilities during app development. A summary of some effective ML/DL-based models used for detecting vulnerabilities in Android code is provided in Table 3.3. This table provides a comparison of the methodology, analysis technique, ML/DL methods or frameworks used, tools and datasets utilised, and the overall accuracy of the model.

Table 3.3: ML/DL-based Android Vulnerability Detection Mechanisms

Study	**Summary of the Methodology**	**Analysis Technique**	**Used ML/DL Methods**	**Used Datasets/ Tools/ Methods**	**Accuracy of the Model**
[117]	The PE data extraction module and the image generation module are utilised to produce input data for each respective module. Subsequently, each model independently determines whether it is malicious by using machine learning algorithms. These algorithms take as input the images that are generated from the image generation module.	Static Analysis	CNN	The dataset is created from Windows portable program files and by utilising the Microsoft Malware Classification Challenge dataset [140].	98.77%

Continued on next page

Table 3.3: ML/DL-based Android Vulnerability Detection Mechanisms (Continued)

Study	**Summary of the Methodology**	**Analysis Technique**	**Used ML/DL Methods**	**Used Datasets/ Tools/ Methods**	**Accuracy of the Model**
[118]	An analysis based on ML is conducted to distinguish between vulnerable and non-vulnerable source code. This is achieved by extracting the AST of a given source code fragment and converting it into a numerical array representation. This process ensures that the structural and semantic information contained in the source code is preserved.	Hybrid Analysis	MLP and a customised model	The publicly available Draper VDISC Dataset [6] is utilised. The proposed model is evaluated by comparing it with the code2vec method [141].	70.1%
[125]	A collection of function calls, consisting of 9,872 sequences, is gathered as features to illustrate the patterns of binary programs while they are running. Subsequently, deep learning models are utilised to forecast the vulnerabilities of these binary programs, based on the data collected.	Dynamic Analysis	CNN, LSTM, and CNN-LSTM	The dataset is created from a sequence of 9,872 function calls. The VDiscover tool [142] is also utilised.	83.6%

Continued on next page

Table 3.3: ML/DL-based Android Vulnerability Detection Mechanisms (Continued)

Study	**Summary of the Methodology**	**Analysis Technique**	**Used ML/DL Methods**	**Used Datasets/ Tools/ Methods**	**Accuracy of the Model**
[126]	An algorithm using a deep neural network is applied to features that are derived from mining source code. These features are generated using an N-gram model. The neural network uses rectified linear units (ReLU) and is trained using the stochastic gradient descent method along with batch normalisation.	Static Analysis	DNN	The dataset is created by downloading APK files from F-Droid [143].	92.87%
[127]	The most effective, human-readable rules for detecting vulnerabilities are created after choosing the optimal machine learning algorithm to identify Lawofdemeter, BeanMemberShouldSerialize, and LocalVariablecouldBeFinal vulnerabilities. A ten-fold cross-validation was carried out, and the results were examined using performance metrics.	Static Analysis	J48 and JRip	The dataset is created from the Android Universal Image Loader project [144] and the JHotDraw project [145]. The PMD tool is employed to analyse the source code.	96%

Continued on next page

Table 3.3: ML/DL-based Android Vulnerability Detection Mechanisms (Continued)

Study	**Summary of the Methodology**	**Analysis Technique**	**Used ML/DL Methods**	**Used Datasets/ Tools/ Methods**	**Accuracy of the Model**
[130]	APK files are decompiled, and a static analysis is conducted on the manifest file to extract the components and permissions. Following this, the system status is retrieved, and fuzzy testing is carried out through dynamic analysis. Finally, machine learning algorithms are implemented to identify intent-based security issues.	Hybrid Analysis	AB and DT	The dataset is created by downloading 300 APK files from leading app stores.	77%
[131]	APK files are decompiled, and certain features are chosen for static analysis. These APKs are then run in an emulator, and log files are produced from system calls for the purpose of dynamic analysis. Following this, a vector space is created, and machine learning algorithms are implemented as parallel classifiers.	Hybrid Analysis	MLP, SVM, PART, and RIDOR	The dataset is created by downloading APK files from Google Play [146], Wandoujia [147], AMD [139], and Androzoo [148].	98.37%

3.4.2 Using Conventional Methods

Traditional approaches, such as heuristic-based methods, formal methods, and other methods that do not rely on machine learning, can also be employed for detecting code vulnerabilities using various analysis techniques. The studies pertaining to these methods are examined in this section.

Conventional Methods with Static Analysis

Numerous research efforts have employed traditional methods of static analysis to identify vulnerabilities in code. A formal model was developed in Alloy (a language grounded in first-order relational logic) to detect security issues in the Android permission protocol, as presented in [149]. This model automatically analyses the protocol using static analysis techniques to pinpoint potential flaws. It recognises three categories of vulnerabilities in the protocol: URI permission vulnerability, improper delegation vulnerability, and custom permission vulnerability. The model is also capable of addressing the dynamic permission process and reveals that the most commonly used permission is signature-based. Among the four content types in Android, the receiver has been found to occur most frequently. An empirical study was carried out to verify the correlation between potential flaws and security vulnerability. The scalability of the formal analysis approach was also evaluated. With minor configurations, this model can be adapted to other mobile operating systems. By refining the model, it can overcome the constraint of only detecting a limited number of vulnerabilities.

A different static analysis method for detecting vulnerabilities in Android applications was introduced as a vulnerability parser model in [150]. This model is composed of several subcomponents: an APK decompressor, a Manifest.xml parser, a vulnerability vector, and a DexParser. The process begins with the decompression of the APK using a Python script. The Manifest file is then parsed to decompress and decompile the APK file with the help of the Manifest parser. This parser can convert the Manifest into a format that is easy to understand and focuses on security aspects. The DEX parser is employed to parse the decompressed source files. The vulnerability vector identifies vulnerabilities related to file access and exported components. The detection results are classified into four categories: critical, warning, notice, and advice. It would be beneficial if this model could be expanded to consider more categories of vulnerabilities, as its current scope is somewhat limited.

Issues in third-party libraries can also pose threats to the application, making it even more difficult to detect them. ATVHunter, proposed in [151], is a model designed for the reliable detection of third-party library versions. This model offers comprehensive information on libraries and vulnerabilities by identifying the vulnerability in library versions and extracting control flow graphs and opcodes. The dataset used included 189,545 unique third-party libraries with 3,006,676 versions and encompassed 1180 common-vulnerable enumerations. An additional 224 security bugs were created to analyse this model. The detection process involved several steps including pre-processing, module decoupling, feature generation, library identification, and identification of vulnerable library versions. ATVHunter detects vulnerabilities with a precision of 98.58% and a recall of 88.79% at the library level, and a precision of 90.55% and a recall of 87.16% at the version level. The study's limitations include its focus solely on Java libraries, the use of only static analysis, the detection of only known vulnerabilities, and the use of only free apps for the study. These limitations present opportunities for further enhancement of the study.

Android web view objects can also be a source of vulnerabilities. A method for detecting these, called the WebVSec framework, was proposed using a static analysis approach in [152]. This study primarily focused on four kinds of vulnerabilities: Interface to Interface vulnerabilities, Interface to WebViewClient vulnerabilities, WebViewClient to WebViewClient vulnerabilities, and Reverse vulnerabilities. The framework was built on the Androguard tool and used AndroZoo as the dataset for analysis. The WebVSec framework involves five key steps: decompilation, identification of interface and WebViewClient classes, identification of methods, method abstraction, and path analysis to detect the aforementioned four vulnerabilities. The experiments analysed 2,000 Android apps and found 48 applications with the four types of vulnerabilities. On average, an application can be analysed in 49 seconds. However, the framework could still benefit from enhancements, such as analysing WebView vulnerabilities generated through Java codes, as it currently only considers Javascript.

The DroidRA model, as proposed in [153], is a designed and implemented approach that seeks to enhance existing static analysis for Android, by managing reflection in apps. It has the ability to determine the targets of reflective calls through a constraint-solving mechanism by instrumenting Android apps to supplement reflective calls with their explicit standard Java calls. The analysis is facilitated by three modules: a Jimple pre-processing module, a reflection analysis module, and a booster module. The model took into account a random selection of 100 real-world apps that contain reflective calls

and at least one sensitive data leak to validate the results of the static analysis. A key advantage of this model is its ability to reveal dangerous code, such as sensitive data leaks and sensitive API calls, which are not visible in other static analysis-based analysis mechanisms. However, a limitation identified is that the single entry-point method may not encompass all the reflective calls, which warrants further exploration. It would be intriguing for future studies to apply these boosting mechanisms to other static analysis techniques used in detecting Android vulnerabilities.

Conventional Methods with Dynamic Analysis

A limited number of studies have incorporated dynamic analysis along with traditional methods. The research in [154] discussed Android app vulnerability detection, drawing inspiration from a case study of web function vulnerabilities. Categories of Android apps, including browsers, shopping, and finance, were scrutinised for security by downloading and examining 6,177 apps. The study analysed four vulnerabilities: Alibaba Cloud OSS credential disclosure vulnerability, improper certificate validation, Web-View remote code execution vulnerability, and Web-View bypass certificate validation vulnerability (sourced from the China National Vulnerability Database [155], CVE list [35], and CWE list [156]). The proposed method, named VulArcher, utilised a heuristic vulnerability search algorithm to verify the accuracy of the analysis. Inputs for this algorithm included all sensitive APIs and methods that could lead to vulnerabilities in the app, a set of rules for fixing vulnerabilities, and a set of rules that trigger the vulnerability. The algorithm outputs detailed code snippets of the vulnerability and the path where vulnerabilities are located. A key feature of the proposed model is its ability to detect vulnerabilities in both packed and unpacked apps. The model includes steps for decompilation, packer identification, unpacking (if packed), building a taint path, and detection. This model can operate with high average accuracy, achieving a detection rate of 91% and demonstrating high efficiency, low computational cost, and high scalability. Identified limitations of this study include the use of an outdated dataset and the integration of third-party tools, which could be revised for improved accuracy in detecting newer vulnerabilities.

Another tool for detecting Android vulnerabilities, named VScanner, was proposed in [29], which is based on dynamic analysis and can identify all known system-level vulnerabilities. The framework of this tool is built on a scalable Lua script engine, a lightweight scripting language. The VScanner uses exploitation for dynamic detection and feature matching for static detection. It can identify vulnerabilities with high efficiency and a low false alarm rate (nearly 100% detection accuracy) using 18 implemented plugins.

Due to the high scalability of the proposed system, it is easy to add new vulnerability triggers. When a vulnerability is triggered via an API call, code execution, or database exploit, a feature matching database is used with scan components (information collection and feedback) in the Lua engine to provide reports and logs. This research proposed a vulnerability taxonomy by Proof of Concept and Attack Surface (POCAS) as existing taxonomies are still immature and specific to Android. In POCAS, vulnerabilities were divided into native layer vulnerabilities (i.e., memory corruption, permission management, kernel escalation, input validation) and Java layer vulnerabilities (i.e., component exposure, file management, information disclosure, logic error). The model was applied for two case studies, namely FakeSMS and CVE-2014-1484 from the National Vulnerability Database [43]. VScanner was tested in fifteen Google simulators, five Android smartphones, eight Genymotion emulators, and seven third-party customised Android systems, and it provided high accuracy and efficient results. The quality of the proposed framework can be improved by increasing the number of plugins used for vulnerability detection and optimising the structures to enhance efficiency.

Conventional Methods with Hybrid Analysis

A number of traditional methods have employed hybrid analysis techniques to identify vulnerabilities. The empirical research conducted in [157] used hybrid analysis to detect eight common vulnerabilities in Android, drawing from a random selection of twenty-nine apps from the EATL app store [158] and six apps from the Google Play store. These eight common vulnerabilities pertained to storage access, web views, SQLite database encryption, intents, advertisement module, outdated or sensitive APIs, short messages, phone calls, and Android debug mode. The study utilised three quality tools: AndroBugs [159], SandDroid [160], and Qark [161] to test and reveal these vulnerabilities. The study also discussed countermeasures for these vulnerabilities, such as more secure use of web views, storing essential files and backups in internal storage rather than external storage, and disabling the debug mode when releasing the apps. The study could benefit from further analysis of apps and additional vulnerabilities by expanding the sample size.

The method for mining application vulnerabilities proposed in [162] employs a hybrid approach, initially conducting static analysis followed by dynamic analysis. This model enhances mining accuracy by integrating fuzzy dynamic testing technology with static analysis while performing reverse analysis on the application. In the static analysis phase, APK files are decompiled to obtain the source files using Dex2Jar and JD-GUI tools and libraries [103]. Subsequently, the feature extraction process creates a feature

vector of API functions, privileges, components, and library files. The scan engine, which comprises data flow analysis, regular expressions matching, and file detection using a vulnerability rule base, is used to obtain the analysis results. Fuzzy testing is employed to carry out dynamic analysis in a natural machine environment with taint analysis. This is done after the static analysis by running the application with test cases, semi-effective data, execution data, taint tracking, and monitoring exceptions. This model can detect vulnerabilities with a detection rate exceeding 95%, which can be further optimised by increasing the number of detectable vulnerabilities through enhanced analysis techniques.

AndroShield [100] proposed another approach based on hybrid analysis, with a focus on detecting vulnerabilities in Android applications. This model was tested against a variety of applications for different security flaws. It is capable of detecting information leaks, insecure network requests, and common flaws that could harm users, such as intent crashes and exported Android components. The proposed model features a three-layer architecture (application, presentation, data) and employs a methodology that includes APK reverse engineering, decoding of the manifest file, extraction of meta-data, performance of static and dynamic analyses, and generation of reports. It can also produce a detailed report that includes the overall risk level of the application and the identified vulnerabilities. However, some limitations have been identified in this publicly available framework, such as its inability to detect deprecated and vulnerable libraries, its lack of analysis of native libraries, and its non-applicability to apps written in other programming languages like Kotlin.

A comparative summary of studies pertaining to traditional models used in vulnerability detection methods is presented in Table 3.4. This table encapsulates the vulnerabilities considered, the findings or capabilities, the limitations, the datasets used, the tools employed, and the methods utilised in these works.

Table 3.4: Conventional Methods of Android Vulnerability Detection

Study	**Considered Vulnerabilities**	**Findings/ Capabilities**	**Limitations**	**Used Datasets/ Tools/ Methods**
[29]	Vulnerabilities present in the native layer and Java layer.	It has the capability to identify all recognised system-level vulnerabilities. Additionally, it suggests a classification of vulnerabilities based on Proof of Concept and Attack Surface.	Employing a restricted set of plugins for the detection of vulnerabilities.	Lua Scripts Engine
[100]	Leakage of information, insecure network requests, and crashes caused by intents.	It produces a comprehensive report that includes the overall risk level of the application and the vulnerabilities that have been identified.	Incapable of detecting vulnerable libraries and analysing native libraries.	ApkAnalyzer [163], FlowDroid [164]

Continued on next page

Table 3.4: Conventional Methods of Android Vulnerability Detection (Continued)

Study	**Considered Vulnerabilities**	**Findings/ Capabilities**	**Limitations**	**Used Datasets/ Tools/ Methods**
[112]	Vulnerabilities associated with SSL/TLS Certificates.	It determines the potential security threats in apps when SSL/TLS is implemented using static analysis. It also ascertains the susceptibility of apps to man-in-the-middle and phishing attacks.	The verification of apps with intricate method implementations is not possible, which results in false negatives.	APK dataset obtained from the Qihoo 360app [165] app market and Google Play [146]
[149]	URI permissions, improper delegation issues, and custom permissions.	It recognises that the permission based on signature is the most commonly used, and among the four content types in Android, the receiver occurs with the highest frequency.	The detection is limited to only a handful of vulnerabilities.	Alloy [166]

Continued on next page

Table 3.4: Conventional Methods of Android Vulnerability Detection (Continued)

Study	**Considered Vulnerabilities**	**Findings/ Capabilities**	**Limitations**	**Used Datasets/ Tools/ Methods**
[152]	Interface to Interface, Interface to WebViewClient, WebViewClient to WebViewClient, and Reverse vulnerability.	It has the capability to analyse an application in less than a minute, specifically 49 seconds.	It is not capable of analysing the WebView vulnerabilities that are produced through means other than Javascript.	BabelView [167]
[153]	Vulnerabilities arising from leaks of sensitive data and API calls.	A significant number of Android apps depend on reflective calls, which are typically employed following certain common patterns.	It may not be possible to reveal all reflective calls due to the utilisation of a single entry-point method.	Google Play [146], AndroZoo [148]
[154]	Disclosure of Alibaba Cloud OSS credentials, improper validation of certificates, execution of remote code in Web-View, and bypassing certificate validation in Web-View.	It has the capability to identify vulnerabilities in both packed and unpacked applications with minimal computational cost. The model also exhibits high average accuracy, detection rate, efficiency, and scalability.	Utilising an outdated set of APKs and incorporating third-party tools.	APK dataset obtained from Wandoujia [147], Qihoo 360app [165] and Huawei [168] App Stores

Continued on next page

Table 3.4: Conventional Methods of Android Vulnerability Detection (Continued)

Study	**Considered Vulnerabilities**	**Findings/ Capabilities**	**Limitations**	**Used Datasets/ Tools/ Methods**
[157]	Vulnerabilities related to storage access, web views, SQLite database encryption, intents, analysis of advertisement modules, outdated or sensitive APIs, short messages and phone calls, and Android debug mode.	It deliberates on the remedial measures for the identified vulnerabilities.	Sample size is limited.	AndroBugs [159], SandDroid [160], Qark [161]

3.4.3 Prevention Techniques

Addressing code vulnerabilities during the initial stages of app development is more beneficial than identifying them post-development. Hence, preventive measures can be incorporated as frameworks, tools, and plugins into the development environments, providing additional support to app developers with automated methods for detecting vulnerabilities. The analysis of experimental outcomes in [169] highlighted the necessity for automated support for detecting code vulnerabilities when creating secure applications that perform well. Android developers were participants who had to suggest suitable fixes for given vulnerable code samples, such as SQL injections, encryption problems, and hard-coded credentials. Furthermore, the 'stitch in time' approach proposed in [37] outlined methods for detecting vulnerabilities in Android apps during development. Developers can input source code and proceed with the development process while the model checks for known security-related issues. If such issues exist, developers are notified accordingly. As a result, developers have the advantage of creating less vulnerable source

code. However, this method only uses known vulnerabilities. Therefore, ML/DL-based methods could be employed to adapt to the evolving nature of issues related to source code. The model could be further refined to learn from user errors and bugs.

In addition to issuing an alert, it is highly beneficial to inform the app developer about the severity level of detected vulnerabilities. Android Lint is a useful tool for identifying vulnerabilities in a given Android source code using static analysis, as discussed in [170]. It can spot 339 issues related to security, performance, correctness, usability, internationalisation, and accessibility. Android Lint employs either an AST or a Universal AST generated from the source code. Other available Linters include Infer, PMD, FindBugs, CheckStyle, Detekt, and Ktlint, as mentioned in [171]. The OASSIS study [172] proposed a method to rank warnings generated from Android Lint using static analysis. This method utilised app user reviews and sentiment analysis to pinpoint app problems. Due to the prioritised warnings, developers can take appropriate action to rectify the vulnerability issues.

The model presented in [173] introduced a method named MagpieBridge, which integrates static analysis with development environments. Although this plugin can be integrated with code editors like Eclipse, IntelliJ, PyCharm, Jupyter, and Sublime Text, its integration with Android Studio was not discussed. On the contrary, the DevKnox plugin [174] for Android Studio is capable of detecting and resolving security issues during the code-writing process for Android applications. FixDroid [37] can be employed to receive security-focused suggestions and fixes to address vulnerabilities during the development of Android applications. It can also be integrated with Android Studio, and its functionality can be further enhanced by incorporating ML to offer suggestions.

In the study [175], a new framework called SOURCERER was introduced. This framework guides app developers in detecting, prioritising, and mitigating vulnerabilities by following secure development guidelines, using static analysis techniques. The application of this framework provides developers with a brief list of vulnerabilities. SOURCERER operates in three stages: identifying assets, mapping vulnerabilities to assets, and mitigation. The authors validated this framework by testing it on 36 financial apps for Android, with three developers participating in the experiment. The findings revealed that developers, on average, spent 15 minutes on asset identification, 30 minutes on detecting and prioritising vulnerabilities, and 20 minutes on finding mitigations when using this framework. Importantly, the use of SOURCERER did not add complexity to the security testing process of Android apps. However, the performance of this framework

could be influenced by several factors, including the limited number of apps used as samples, the limited involvement of developers in the experiments, and the developers' prior knowledge. To address some of these limitations, the authors suggest the possibility of introducing an automated process.

The VuRLE tool [176], is designed to automatically detect and fix vulnerabilities in code, aiding developers in handling some vulnerabilities. The initial step involves training the model and grouping similar edit blocks using a set of repair examples. For each group, repair templates are created to identify vulnerable groups by applying transformative edits. This process involves traversing a generated AST and using 10-fold cross-validation. The model was able to repair 101 out of 183 identified vulnerabilities from 48 real-world applications (including Android, web, word-processing, and multimedia apps) written in Java. However, some vulnerabilities could not be repaired due to issues such as unsuccessful placeholder resolution, a lack of repair examples, and partial repairs. The repair rate of this tool was 65.69%, which is relatively low. However, this rate could potentially be improved by training the model with a larger set of vulnerable code samples.

3.5 Use of XAI

In code vulnerability detection and rectification, it is crucial to differentiate between factual and counterfactual explanations. A factual explanation answers the *what* or *why* questions by providing empirical evidence that supports a specific AI model outcome based on the given input. This explanation also helps to locate vulnerabilities within the code. Conversely, a counterfactual explanation addresses the *Why-not* or *How-to* questions by constructing a hypothetical scenario that results in a more favourable outcome. This method assists in illustrating how to amend the identified vulnerabilities [177].

In [178], the authors detail the development of a human-in-the-loop XAI system specifically engineered to alleviate vulnerabilities. This system elucidates model predictions to forensic experts via feature attributions, equipping them with the necessary insights to make essential corrections. Additionally, the research presented in [179] introduces the DisCERN counterfactual explainer as a valuable tool for rectifying code vulnerabilities. This tool leverages insights from feature attribution explainers and pattern matching to propose correction recommendations.

Once an AI-driven prediction has been produced, the probability of predictions in binary or multi-class classification models can be determined using various Python frameworks.

Some commonly used frameworks include Shapash, Dalex, Explain Like I am 5 (ELI5), Local Interpretable Model-agnostic Explanations (LIME), Shapley Additive Explanations (SHAP), and Explainable Boosting Machines (EBM), among others, as outlined in [180]. The selection of a framework depends on the specific needs of the prediction task. Therefore, it is worth exploring the potential applications of these XAI techniques in AI-powered models for detecting vulnerabilities in Android code.

Several XAI techniques are particularly valuable for AI models focused on detecting code vulnerabilities. Feature Importance Visualisation, such as permutation importance or SHAP, is instrumental in identifying the code patterns that significantly influence the model's decisions. This capability enables users to pinpoint which code sections contribute most to vulnerability classifications. Attention Mechanisms are also crucial, especially in models employing them (common in natural language processing and increasingly in code analysis). These mechanisms visualise attention weights, highlighting the critical parts of a code snippet that influence the model's decisions. This visualisation helps users to concentrate on areas potentially vulnerable to security issues. Counterfactual Explanations are another essential technique. These involve creating alternative scenarios to demonstrate how changes in the code would affect vulnerability classifications. By modifying parts of the code and observing the resulting changes in classification, users gain insights into the triggers for vulnerabilities. Additionally, Interpretable Models like decision trees or rule-based systems are valuable for explaining why specific classifications are made based on predefined rules or thresholds in code vulnerability detection. Interactive Visual Interfaces further enhance user understanding by providing tools for visualising model predictions and explanations. For example, interactive dashboards allow users to explore different sections of the code and view real-time explanations, improving overall usability [178, 181, 182].

3.6 Existing Tools for Analysing Apps and Detecting Vulnerabilities

Tools and frameworks for application and source code analysis are advantageous in executing various analysis procedures. After the completion of these analyses, tools designed to detect vulnerabilities can be utilised to pinpoint susceptible parts of the source code. The significance of automated code analysis tools in identifying vulnerabilities has been underscored in the surveys and interviews pertaining to long-term software security interventions, as referenced in [183].

The research [184] employed six primary attributes to evaluate security analysis tools. These include 1) tool versus framework, 2) free versus commercial, 3) maintained versus unmaintained, 4) vulnerability detection versus malicious behaviour detection, 5) static analysis versus dynamic analysis, and 6) local versus remote. This research analysed sixty-four solutions, taking into account supported Android versions, multiple operational modes, supported API levels, relevant categories of vulnerabilities, the presence of vulnerabilities, and the inputs required by the tools.

Another investigation in [184] assessed 64 tools and empirically tested 14 vulnerability detection tools for 42 known vulnerabilities identified in [185]. Some of these widely used tools are FixDroid, AndroBugs, Qark, and Flowdroid as compared in Table 3.5. It was discovered that only 30 out of the 42 vulnerabilities could be detected. These 42 known vulnerabilities were divided into seven categories, namely, a) Cryptography - 4 vulnerabilities, b) Inter-Component Communication (ICC) - 16 vulnerabilities, c) Networking - 2 vulnerabilities, d) Permission - 1 vulnerability, e) Storage - 6 vulnerabilities, f) System - 4 vulnerabilities, and g) Web - 9 vulnerabilities and some of these can be mapped with vulnerability repositories such as CWE. This study utilised AndroZoo [148], a source of real-world Android applications comprising approximately 5.8 million APKs.

The empirical study carried out in [186] pinpointed the correlation between static software metrics and the most informative metrics that can be employed to detect code vulnerabilities in Android source code. The AndRev tool, introduced in [187], utilised static analysis to extract permissions by reverse engineering APKs with a tool scripted in batch. The extracted features were stored in a feature vector and analysed to discern permission patterns, taking into account the app category. This tool attempted to eliminate unnecessary permissions for the app through reverse engineering and rebuilding. A security analysis was also conducted to identify vulnerabilities by incorporating a tool named Quixxi. The study revealed that medium-risk vulnerabilities outnumbered low and high-risk vulnerabilities. The accuracy of the tool could be further verified using a larger dataset, as this study utilised a limited dataset of 50 apps for the preliminary analysis.

Table 3.5 presents a comparison of various tools and frameworks that can be run on local machines for the purpose of analysing applications, scrutinising source code, and identifying vulnerabilities in Android. The table contrasts these tools and frameworks in terms of their capabilities, limitations, methods of analysis, and usage. Given the numerous limitations associated with these tools, this research proposes a solution that

addresses them by securely detecting real-time Android code vulnerabilities within the Android Studio development environment referring to CWE repository. Additionally, the approach emphasises continuously improving detection capabilities by enhancing the training model.

Table 3.5: Existing Tools for Analysing Apps and Detecting Vulnerabilities

Framework Name/ Tool	**Capabilities**	**Limitations**	**Analysis Technique**	**Usage**
FixDroid [37]	Capability to offer suggestions and solutions oriented towards security to address vulnerabilities.	Dependent on a comparatively small dataset and does not concentrate on enhancing data flow analysis beyond utilising the existing features of IntelliJ IDEA.	Static Analysis	Academic
APKTool [110]	The capability to disassemble the APK using a static analysis tool [188], along with the proficiency to reverse-engineer Android applications by decoding them to a form that closely resembles the original, and then reconstruct the application after making alterations.	Struggles to disassemble and examine APKs that are extensively obfuscated.	Static Analysis	Industrial

Continued on next page

Table 3.5: Existing Tools for Analysing Apps and Detecting Vulnerabilities (Continued)

Framework Name/ Tool	**Capabilities**	**Limitations**	**Analysis Technique**	**Usage**
AndroBugs [159]	Capability to identify potential Android security vulnerabilities, and scrutinise the code for adherence to security best practices and the presence of hazardous shell commands.	Incapable of offering a comprehensive and detailed explanation to assist in resolving any potential security issues.	Static Analysis	Industrial
QARK [161]	Capability to identify vulnerabilities related to security in Android applications, whether in APKs or source code.	Incapable of analysing apps that are heavily obfuscated and necessitates significant CPU usage during the decompiling process.	Static Analysis	Industrial
FlowDroid [164]	Capability to calculate data flows statically.	Presumes that all contents continue to be tainted, even if a single array element is overwritten by an untainted value.	Static Analysis	Academic
DevKnox [174]	Capability to identify and rectify security issues during the process of code writing.	Incapable of identifying new vulnerabilities and lacks support for the most recent Android environments.	Static Analysis	Industrial

Continued on next page

Table 3.5: Existing Tools for Analysing Apps and Detecting Vulnerabilities (Continued)

Framework Name/ Tool	**Capabilities**	**Limitations**	**Analysis Technique**	**Usage**
MalloDroid [189]	Capability to detect faulty SSL certificate validation using the Androgurd framework [133].	The analysis could be unsuccessful if the app is obfuscated and it is unable to test the complete workflow.	Static Analysis	Academic
COVERT [190]	Capability to conduct compositional analysis of vulnerabilities between apps.	Incapable of detecting vulnerabilities related to native code and permission leakages.	Static Analysis	Academic
HornDroid [191]	Capability to conduct static analysis of information flows, and the ability to abstract the semantics of Android apps soundly to formulate security properties.	It overestimates the lifecycle of fragments by running all the fragments in conjunction with the containing activity in a manner that is insensitive to flow. This could result in precision issues in actual applications.	Static Analysis	Academic
JAADAS [192]	Capability to examine API misuse, taint flows in an inter-procedure style, local-denial-of-services, and intent crashes.	It might encounter crashes when analysing obfuscated applications, and the JSON output file does not effectively illustrate the extent of potential issues.	Static Analysis	Industrial

Continued on next page

Table 3.5: Existing Tools for Analysing Apps and Detecting Vulnerabilities (Continued)

Framework Name/ Tool	**Capabilities**	**Limitations**	**Analysis Technique**	**Usage**
DIALDroid [193]	Capability to detect privilege escalations and collusion between apps.[194].	Incapable of resolving reflective calls if their arguments lack string constants and may encounter difficulties in computing some ICC links due to the disregard of over-approximated regular expressions.	Static Analysis	Academic
MARVIN [195]	Capability to evaluate the potential maliciousness of previously unidentified apps using machine learning techniques, and to generate a precise snapshot of malware behaviour that can be utilised to gauge the risk associated with apps.	Incapable of automatically intercepting apps during the download process from marketplaces; apps must be manually submitted to MARVIN.	Static Analysis	Industrial
MobSF [196]	Capability to conduct static analysis, hybrid analysis, penetration testing, and offer a REST API for integration with development environments.	Incapable of conducting API testing and experiences some problems in the emulator during the execution of apps in hybrid analysis.	Hybrid Analysis	Industrial

Continued on next page

Table 3.5: Existing Tools for Analysing Apps and Detecting Vulnerabilities (Continued)

Framework Name/ Tool	**Capabilities**	**Limitations**	**Analysis Technique**	**Usage**
Amandroid [197]	The capability to analyse the data flow between components for security assessment.	Fails to identify security vulnerabilities where exceptions might arise and struggles with managing reflections and concurrency.	Static Analysis	Academic
Snyk [198]	Claims to have near zero false positives, it can be integrated with IDEs.	Needs to be executed in parallel, cannot provide reasons for detected vulnerabilities, and offers limited features in the free version.	Static Analysis	Industrial
ImmuniWeb Mobile-Suite [199]	Claims to be zero false-positive, it can be integrated with SDLC, provides actionable remediation guidelines, and supports both mobile app and backend testing.	Needs to be executed concurrently with the development environment as a separate tool.	Static Analysis	Industrial

Continued on next page

Table 3.5: Existing Tools for Analysing Apps and Detecting Vulnerabilities (Continued)

Framework Name/ Tool	**Capabilities**	**Limitations**	**Analysis Technique**	**Usage**
Drozer [200]	Interacts with the Dalvik VM and other endpoints of the app to detect vulnerabilities, supports penetration testing, and searches for security flaws in apps, all while being free and open source.	Inability to detect vulnerabilities in real-time, integrate with Android development environments, and is less user-friendly due to its command-line-based approach.	Hybrid Analysis	Industrial
Astra Pentest [201]	Performs over 8000 test cases to identify vulnerabilities, including misconfiguration errors in code or build settings.	Inability detect code-level vulnerabilities at early stages and needs to be run as a separate program.	Dynamic Analysis	Industrial

3.7 Repositories and Datasets for Vulnerability Detection

Datasets and repositories are instrumental in executing various machine learning or traditional vulnerability detection techniques. Numerous datasets, including Drebin [123], Google Play [146], AndroZoo [148], AppChina [202], Tencent [203], YingYongBao [204], Contagio [205], Genome/MalGenome [135], VirusShare [206], IntelSecurity/-MacAfee [207], MassVet [208], AMD [139], APKPure [209], Android Permission Dataset [210], Andrototo tal [211], Wandoujia [147], Kaggle [134], CICMaldroid [212], AZ [138], and Github [122] are available for conducting these experiments.

In [185], Ghera, an open-source benchmark repository, was unveiled. It documented 25 recognised vulnerabilities in Android apps. Additionally, it outlined some common attributes of vulnerability benchmarks and repositories. The primary objective of this research was to discover Android-specific vulnerability benchmarks to assess tools that could aid app developers. It was found that there were no test suites or benchmarks

to reasonably evaluate vulnerability detection methods. Many relied on standard data and apps from Google Play. During the review phase, 11 characteristics were identified as vulnerability benchmark traits. These included being 1) agnostic to tools and techniques, 2) authentic, 3) feature-specific, 4) contextual, 5) ready for use, 6) user-friendly, 7) version-specific, 8) well-documented, 9) containing both a vulnerability and a corresponding exploit, 10) open to the community, and 11) comprehensive. This repository houses information on the Android Framework's inter-component communication, storage, system, and web vulnerabilities. However, Ghera did not cover vulnerabilities related to networking and sensors. Despite this, none provided a benchmark dataset specifically for detecting vulnerabilities in Android source code. Therefore, it would be beneficial to broaden the scope of the repositories by including more real-world apps.

The research in [33] pinpointed the CVE details [35] as a data analysis source, offering vulnerability statistics on products, versions, and vendors. These CVE details were compiled using NVD [43]. The evaluation of vulnerabilities took into account the mean impact scores and the number of instances. A dataset for rectifying open-source software code vulnerabilities by pinpointing security-related commits in a given source code was presented in [213]. This dataset was curated manually. The research in [214] established a repository called AndroVul, housing Android security vulnerabilities. It encompasses high-risk shell command vulnerabilities, security code smells, and dangerous permissions. It was formed by examining APKs obtained from AndroZoo [215]. It acts as a standard for identifying Android malware and can be utilised for machine learning experiments to detect malicious code through static analysis. A different dataset, presented in [216], is a commit-level dataset for real-world vulnerabilities. It has scrutinised over 1,800 projects and more than 1,900 vulnerabilities based on CVE from the Android Open Source Project.

For the purpose of analysing source code, fifteen tools were identified, five of which can detect vulnerabilities in Android. The study also found that 19 datasets and repositories are available for source code analysis and vulnerability detection. However, most of these datasets and repositories do not focus solely on Android source code vulnerabilities. Therefore, one key finding is the need for a proper dataset. Furthermore, given the limited number of comprehensive tools available for detecting vulnerabilities, it is also important to consider developing new tools for detecting source code vulnerabilities in Android using such datasets.

3.8 Discussion on Application Analysis and Vulnerability Detection Methods

Upon examining the studies, it was found that static analysis was the preferred method of application analysis in 51% of the studies, while 35% employed hybrid analysis. The remaining 14% utilised dynamic analysis. This distribution is depicted in Figure 3.2. The preference for static analysis could be attributed to its benefits in code-level analysis approaches, as it primarily focuses on code features. Additionally, static analysis is more cost-effective compared to the other two methods. Conversely, dynamic analysis necessitates extra resources like emulators or actual devices to execute the source code, and it may not reveal as many vulnerabilities as static analysis. The requirement for APKs that can be compiled might be another factor contributing to the fewer studies using dynamic analysis for vulnerability detection. Hybrid analysis, combining the features of both static and dynamic analysis, is used in an intermediate percentage of studies.

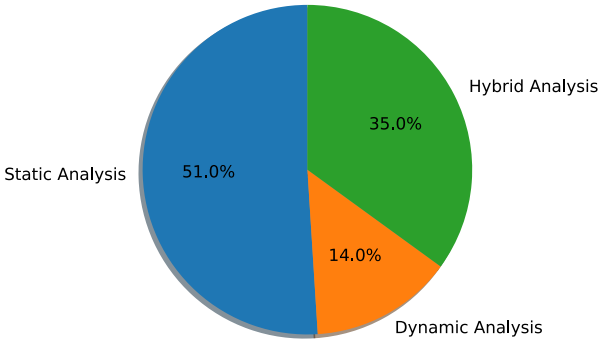


Figure 3.2: Application/source code analysis techniques used in the reviewed studies

The studies reviewed indicate a higher application of machine learning-based methods compared to traditional methods, as shown in Figure 3.3. Prior to 2016, traditional methods were more prevalent in the research community than machine learning methods. However, with the surge in machine learning techniques, researchers began to employ these methods to address problems [5]. The popularity of machine learning and deep learning methods, their ability to deliver highly accurate results, their proficiency in managing complex problems, and their scalability are likely reasons for their extensive use in studies on Android vulnerability detection during the review period.

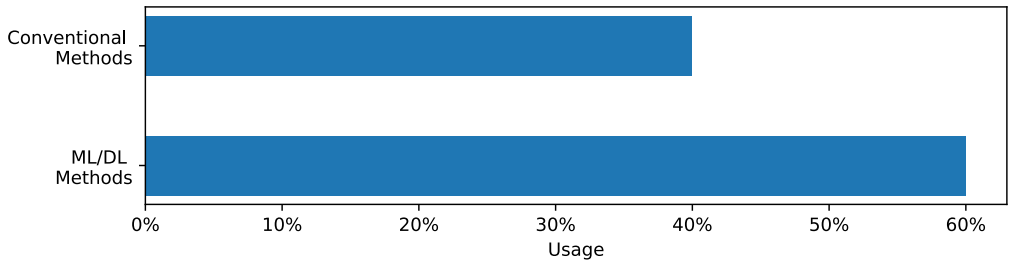


Figure 3.3: Vulnerability detection methods

A significant number of studies on code vulnerability detection have employed the code analysis method for feature extraction. Other commonly used methods include Manifest analysis and system call analysis, as depicted in Figure 3.4.

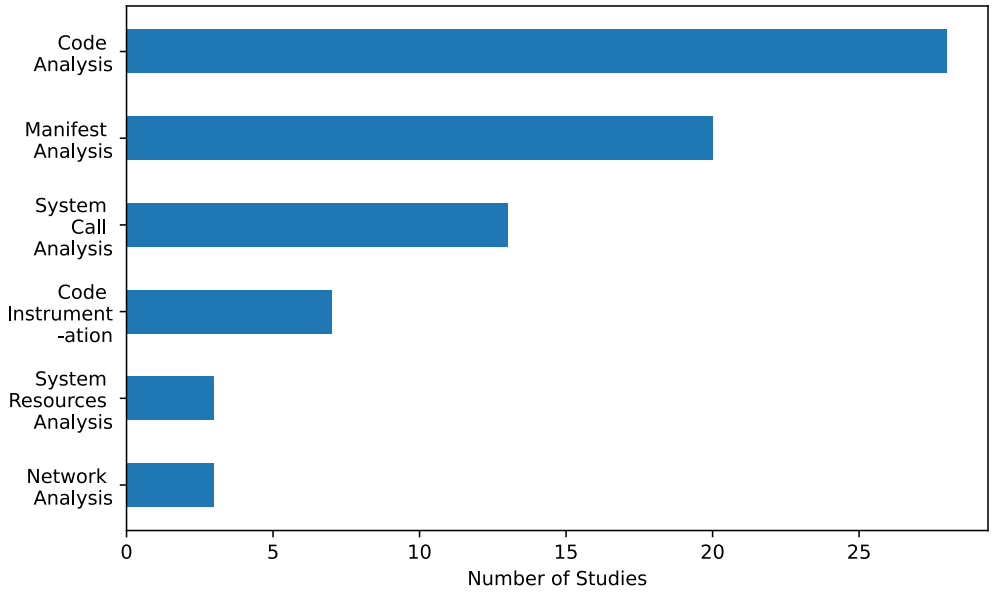


Figure 3.4: Feature extraction methods used in the reviewed studies

The high usage of code analysis could be attributed to its ability to detect a large number of vulnerabilities through source code analysis, rather than through the analysis of permissions or other features. Manifest analysis, which can identify vulnerabilities to a certain degree, such as the types of permissions used in applications, is also frequently used. The detection of vulnerabilities can be based on the permissions required by the application, especially if they are of a dangerous level. This could explain the

relatively high number of studies conducted using this method. System call analysis is used in a substantial number of studies, as it allows for the detection of vulnerabilities to a certain extent by analysing system calls. Methods such as code instrumentation, system resources analysis, and network analysis were used less frequently, as detecting vulnerabilities through these analyses can be challenging.

API calls emerged as the most frequently extracted feature for the analysis and detection of vulnerabilities in Android source code, as shown in Figure 3.5.

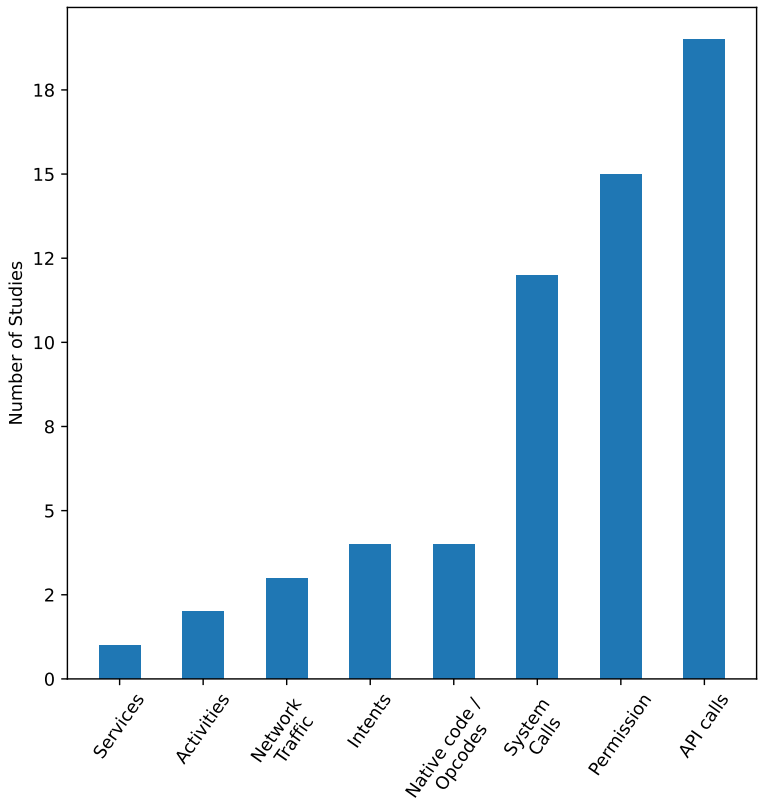


Figure 3.5: Extracted features in the reviewed studies

This feature was predominantly extracted by many static and hybrid analysis methods. The extensive use of API calls for feature extraction could be due to the comprehensive insights they provide into vulnerabilities. Permissions were the second most extracted feature in the studies reviewed. They are the primary feature extracted during manifest analysis, which could account for their high extraction rate given the widespread use of

manifest analysis. System calls also featured prominently as they can reveal numerous vulnerabilities upon analysis. Other features such as native code or opcodes, intents, network traffic, activities, and services were also extracted, albeit less frequently.

The review reveals that only a small number of studies have taken into account prevention mechanisms, such as tools and plugins, which can aid in mitigating vulnerabilities in Android source code. As shown in Figure 3.6, many studies focused solely on detection. For Android application developers, having access to effective mechanisms that employ various advanced techniques to prevent vulnerabilities would be beneficial. Consequently, one of the key findings of this review is the identified need for the development of such a prevention mechanism.

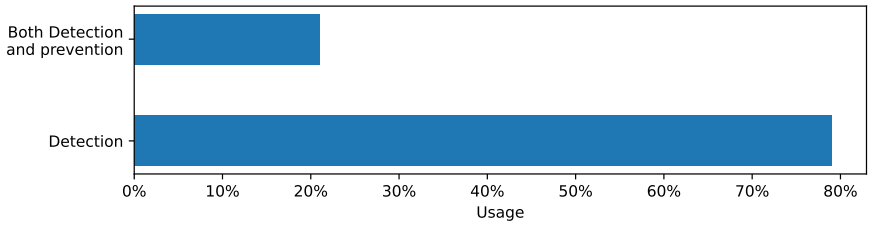


Figure 3.6: Availability of detection and prevention methods

3.9 Chapter Summary

This chapter's primary aim was to pinpoint current methods, grounded in static, dynamic, and hybrid analysis, for detecting vulnerabilities in mobile application source code. The objective was to recognise and evaluate existing methods for identifying vulnerabilities in Android source code. The research question RQ1 was answered while achieving RO1 as specified in chapter 1. Consequently, the literature review chapter made a substantial contribution by providing a thorough and systematic review of vulnerability detection in Android source code. This review highlights the research areas that have been overlooked and scrutinises how current studies address the intricate issues related to vulnerability detection. Additionally, two publications were produced from this chapter as systematic literature review journal papers KD1 [16] and KD2 [17].

Chapter 4

Methodology

This chapter provides a comprehensive overview of the research methodology. It begins with the methodology of the labelled vulnerability dataset creation and then outlines the process of creating a highly accurate, privacy-focused, community-driven AI model. The chapter also outlines how this AI-based model is integrated with Android Studio.

4.1 Overall Approach

The project was structured and the strategy was devised with the aim of creating a precise, privacy-protected, community-driven innovative AI-based method for detecting vulnerabilities in Android source code in real time.

With a thorough examination of the existing literature to pinpoint an under-researched area of study, the project was initiated. This exhaustive investigation encompassed current methods for application and code analysis, potential uses of machine learning and deep learning methodologies, and the presence of datasets. The possibilities of implementing explainable AI, federated learning, differential privacy, and blockchain technology were also evaluated. Owing to the constraints of the available datasets, a unique dataset named LVDAndro was formulated. This dataset was the result of a hybrid scanning method that amalgamated the strengths of various high-precision Android app vulnerability scanning tools. More than 15,000 Android apps underwent scanning, and the vulnerable source code lines were subsequently categorised according to the CWE IDs. The utility of the LVDAndro dataset for pinpointing vulnerabilities in Android code was initially demonstrated through a proof-of-concept using AutoML. To augment the

model's capabilities, an ensemble model was assembled, incorporating multiple machine-learning algorithms and this model was named as ACVED. The use of AutoML for the initial proof-of-concept allowed for rapid experimentation with various ML models, ensuring that the most effective algorithms were identified. The decision to implement an ensemble model was driven by the need to enhance model performance by leveraging the strengths of multiple algorithms. This approach is well-known for improving predictive accuracy and robustness, particularly in complex tasks like vulnerability detection.

Alongside the ensemble model, a shallow neural network model was also crafted to further boost the model's performance. The choice of this neural network was informed by the need for a balance between computational efficiency and model complexity, ensuring that the solution could scale effectively while maintaining high accuracy. Model pruning techniques were utilised during this phase. To tackle the issue of limited training data and to involve more clients in the model training process, federated learning was implemented and this model was named as FedREVAN. To further bolster the model's validation process and foster community involvement, a blockchain-based environment was integrated. The final model named Defendroid, in conjunction with XAI, was incorporated into an API. This API was subsequently embedded as the backend of a newly devised plugin for the Android Studio IDE. This integration empowers developers to identify vulnerabilities in real-time as they code in the Android Studio IDE. To evaluate the practicality and efficacy of the newly created plugin in real-world situations, a user survey was carried out among Android application developers. This survey functioned as a case study, yielding valuable insights into the application of the model and plugin in actual Android app development. The developers' feedback was utilised to pinpoint areas for future enhancements to the plugin. The subsequent subsections discuss this comprehensive approach. The overall approach is illustrated in Figure 4.1.

4.2 Developing the Android Code Vulnerability Dataset

A properly labelled dataset was the main requirement to create a base model for the AI-based community-driven privacy-preserved Android code vulnerability detection. Hence, during this phase, the suitability of existing methods for scanning Android vulnerabilities was evaluated, and a few well-performing scanners were pinpointed. These scanners were then utilised to experiment with creating a new labelled dataset of Android code vulnerabilities, thereby fulfilling the research's RO2 as mentioned in chapter 1. This

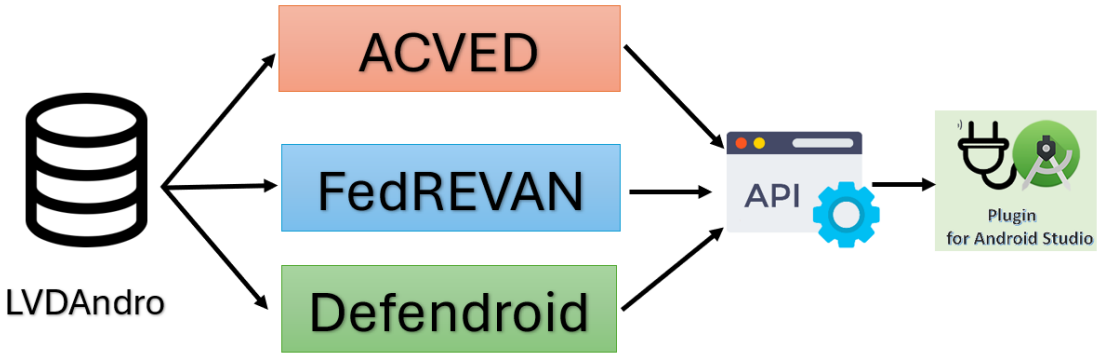


Figure 4.1: Overall Approach

dataset, named LVDAndro, comprises a series of sub-datasets. Each sub-dataset is labelled according to CWE categories and was generated by scanning apps from various sources and in varying quantities.

The detailed process for creating the LVDAndro dataset is illustrated in Figure 4.2. It involves three primary steps: collecting the data, labelling them, and pre-processing.

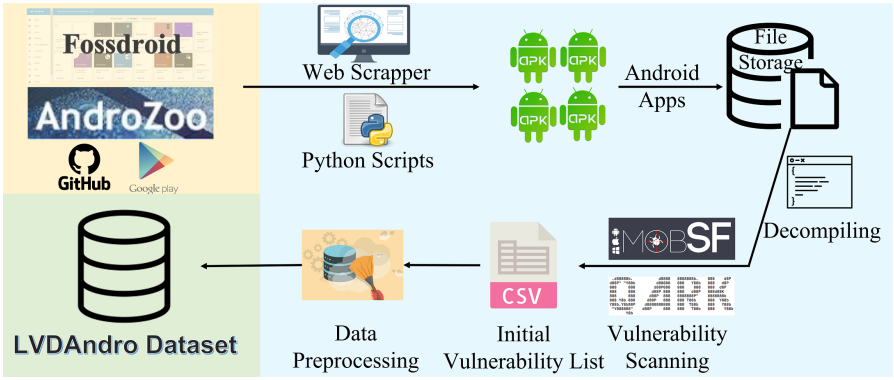


Figure 4.2: The process of generating the LVDAndro dataset

An overview of the LVDAndro dataset including its sub-datasets which are represented under Datasets 01, 02 and 03 is illustrated in Figure 4.3. A detailed discussion of the datasets can be found in Chapter 5.

Android applications were scrapped and downloaded from Fossdroid, Androzoo, GitHub, and Google Play app stores using custom Python scripts. The scraped apps belonged to a variety of categories, including trending apps in education, tools, multimedia, entertainment, connectivity, and health. Scraping can occasionally result in incomplete or

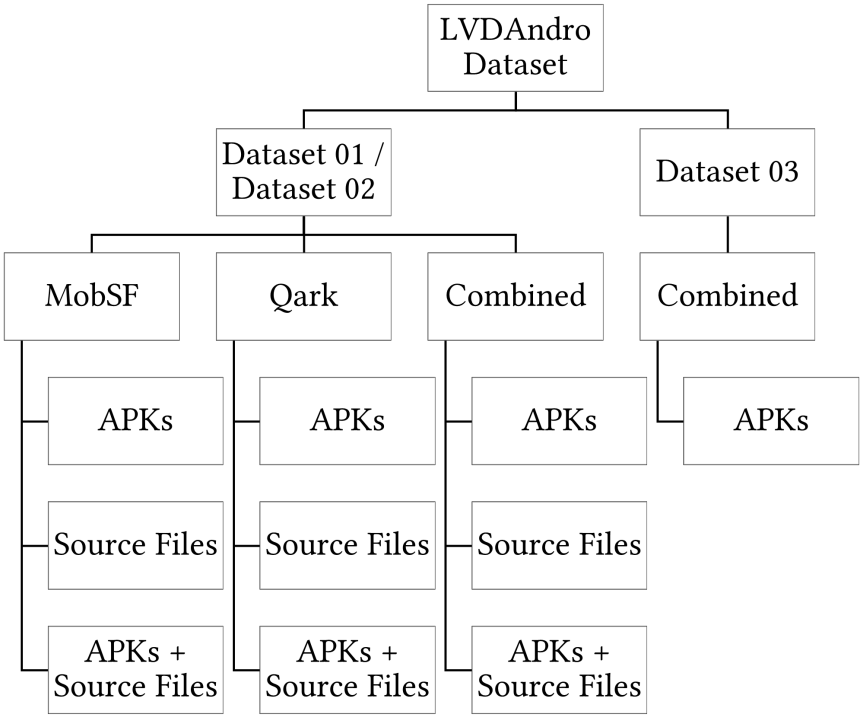


Figure 4.3: Overview of LVDAndro Datasets

corrupted downloads, which may compromise the consistency of the dataset. Ensuring that all apps are fully downloaded and accurately categorised can be particularly challenging, especially in large-scale scraping operations. However, this process was carefully managed and extensively checked using an optimised Python script.

These applications were then stored in a file system and decompiled using another Python script. The decompilation process utilised standard libraries such as JADX, dex2jar, and JD-GUI. These tools may not always produce perfectly decompile code, particularly when dealing with obfuscated or heavily optimised apps. Such inaccuracies in decompilation could lead to missing or misinterpreted code, potentially affecting the subsequent vulnerability analysis. Since many Android applications employ obfuscation techniques to protect their code, decompiling these apps can be particularly challenging and may require additional processing or specialised tools. To address this, the custom-built Python script was designed to automatically discard apps with obfuscated code that could not be decompiled by the scanners used.

Following decompilation, the application files were scanned using a combined approach with the assistance of MobSF and QARK scanners to detect vulnerabilities. These vulnerabilities were then labelled according to the CWE using a specially developed Python script. Subsequently, various pre-processing techniques were applied to standardise the data. The end result of this process was the creation of the LVDAndro dataset. The large scale of the dataset, which includes over 15,000 apps, presents challenges related to data storage, processing time, and resource management. Efficiently managing and processing this volume of data requires robust infrastructure and optimisation techniques. By utilising cloud-based storage solutions, these challenges were effectively addressed.

An in-depth explanation of the dataset creation and its subsequent analysis can be found in chapter 5, and this addressed RQ2 and achieved RO2.

4.3 Developing AI-based Android Code Vulnerability Detection Model

To address RQ3, as outlined in chapter 1, and to meet the established objectives RO3 and RO4, an AI-based model was constructed using the LVDAndro dataset. Initially, the applicability of the dataset for this purpose was validated as a proof-of-concept using an AutoML-based model [217]. The experiments demonstrated the suitability of the LVDAndro dataset for AI-driven models designed to detect vulnerabilities in Android code. Given the satisfactory performance of the AutoML-based model, an ensemble model named ACVED - Android Code Vulnerability Early Detection was suggested, which stacks commonly used ML algorithms for detecting vulnerabilities. The suggested ensemble model attains high combinations of accuracy, precision, recall, and F1-score in both binary and multi-class classifications. This outcome is likely attributable to the ensemble stacking classifier's ability to amalgamate the strengths of all the other classifiers. The ACVED model was further improved by incorporating XAI to elucidate the rationale behind predictions. The fully developed model was then integrated into the backend of a Python Flask-based API, allowing for code lines to be submitted to the API for vulnerability identification. This architecture is depicted in Figure 4.4.

While the ACVED model, based on the ensemble method, demonstrated good performance, it had a limitation in terms of scalability. This was because the model required complete retraining when new data was added to the LVDAndro dataset. Incorporating new data into the LVDAndro dataset is a resource-intensive task, and it can lead to an

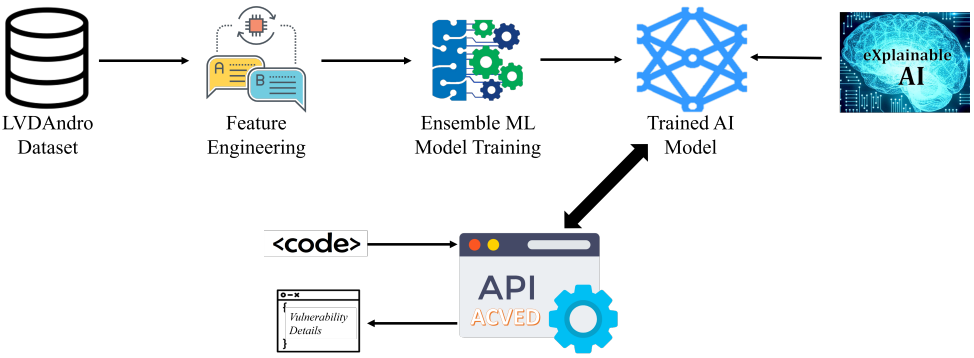


Figure 4.4: API Development Process

increase in the volume of training data, potentially surpassing the capacity of the training machine resources.

To tackle this issue without unnecessarily expanding the size of the dataset, and to introduce more varied training data, without relying solely on the LVDAndro dataset, a federated learning-based model was devised using a neural network architecture. This federated learning environment allows for the integration of multiple clients who contribute the training data to enhance the model. This approach offered a degree of privacy assurance, as it enabled the sharing of model weights within the federated learning environment without the need to share the data among the training clients and the model aggregation server. The ACVED model was then updated with this architecture and it was named as FedREVAN - Real-time Detection of Vulnerable Android Source Code through Federated Neural Network with XAI. This also uses XAI-based suggestions as ACVED. The overall approach of the FedREVAN is depicted in Figure 4.5.

The detailed development of this highly accurate AI model, which is designed to identify vulnerabilities in Android's source code, is discussed in chapter 6. This includes the construction of the AI model that integrates ML, AutoML, DL models, and XAI, all of which are facilitated by the LVDAndro dataset. This addressed RQ3 while achieving the objectives RO3 and RO4.

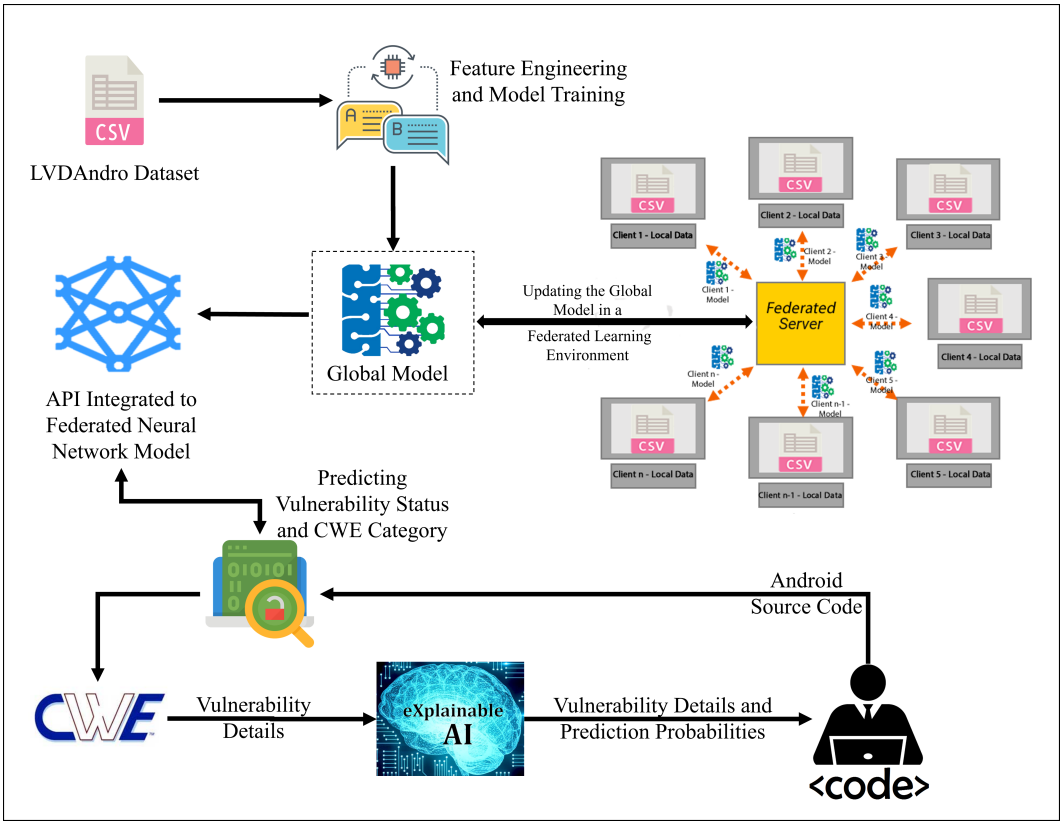


Figure 4.5: Overview of the FedREVAN Model

4.4 Enhancing the Model to a Privacy-Preserved Community Driven Model

The federated learning-based model significantly enhanced the performance of the model. However, to ensure greater privacy, differential privacy was incorporated into this federated learning environment. This approach made it possible to achieve both data privacy and model privacy. As a result, a large number of participants could join the federated learning-based training.

This led to another challenge of finding a substantial number of clients who could contribute to the environment. To address this, the model was then released to a custom-built, community-driven blockchain-based architecture. This allowed anyone to participate in the blockchain-based federated learning environment. A consensus algorithm was implemented accordingly to ensure that participants contribute positively to the model without compromising its current performance level.

One of the significant advantages of using a blockchain-based architecture is the distributed ownership it provides. Unlike traditional server-based models, controlled by a single agent or organisation, a blockchain-based system shares ownership among all participants. This decentralised approach offers several benefits. With no single point of failure, the system is inherently more secure against attacks, as each participant holds a copy of the blockchain, making it extremely difficult for malicious actors to alter the data. All transactions and model updates are recorded on the blockchain, creating a transparent and immutable ledger. This transparency builds trust among participants, as they can independently verify contributions and changes made to the model. Distributed ownership fosters an environment where participants are more invested in the model's success and accuracy, driving higher engagement and quality contributions. To further encourage participation and ensure the model's growth, a reward system for contributions can be implemented in the blockchain-based federated learning environment during future expansion stages.

A similar model architecture as in FedREVAN was used while integrating blockchain and differential privacy with the federated environment, but it was given a new name, *Defendroid*. This integrates all these aspects of model training, explainability, privacy, and community-driven architecture. The integration of blockchain with differential privacy in the Defendroid model ensures that data privacy is maintained while benefiting from a collaborative environment. Differential privacy techniques add noise to the data, preventing individual contributions from being traced back to specific participants and thereby protecting their privacy. The community-driven aspect of Defendroid means that the development and improvement of the model are guided by the collective input of its participants. This democratic approach allows for a more diverse and inclusive model development process, where the community can propose and vote on new features, improvements, and other changes, ensuring that the model evolves to meet the needs of its users. The transition from FedREVAN to Defendroid is illustrated in Figure 4.6. The detailed process of this is discussed in chapter 7.

The model which has the highest prediction results was updated to the vulnerability detection API created earlier. Subsequently, a plugin for Android Studio was created that interfaces with the developed API. When this plugin is enabled within Android Studio, it allows developers to examine code vulnerabilities by submitting code lines to the API through the plugin. After detecting any vulnerabilities, the developers can overcome them through the suggestions provided by the plugin. The process of the plugin is depicted in Figure 4.7. Detailed steps of the plugin usage are discussed in chapter 7.

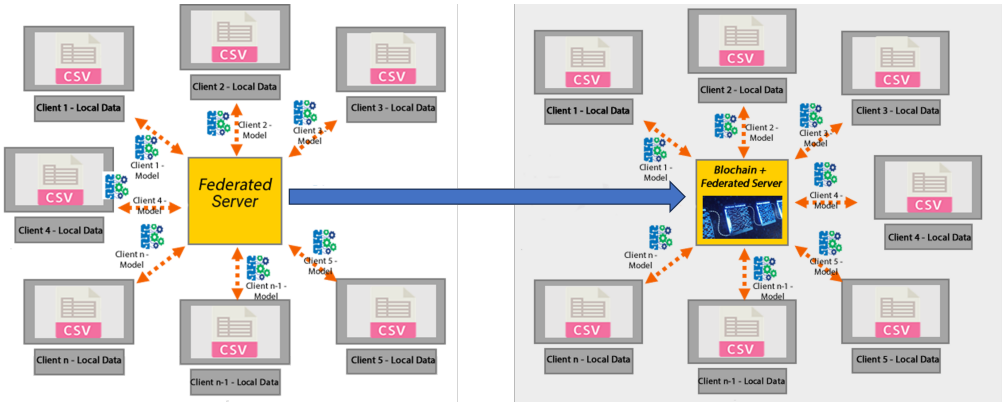


Figure 4.6: Environment Changes from FedREVAN to Defendroid

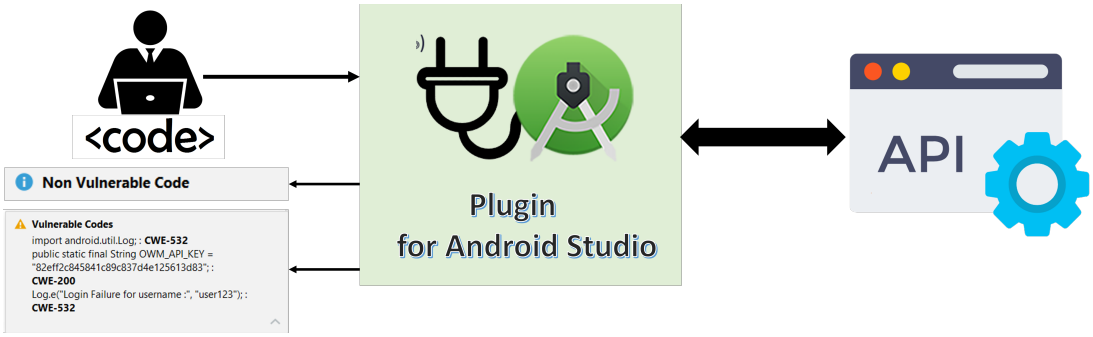


Figure 4.7: Process of Vulnerability Detection Plugin

Using this approach it was possible to address RQ4 while achieving RO5. The performance of the plugin was also assessed with the assistance of 63 Android developers, as discussed in chapter 8.

4.5 Chapter Summary

The main objective of this chapter was to present the approach employed in the thesis. The comprehensive methodology significantly contributed by introducing a unique, community-driven, highly precise, and privacy-focused AI-based technique for real-time detection of vulnerabilities in Android code. The methodology of the thesis successfully addressed all the research queries and met all the research goals as outlined in chapter 1. Therefore, the dissemination of knowledge of the findings and contributions through 3 Journal Papers and 5 Conference Papers in prestigious forums highlights the originality and thoroughness of the thesis.

Chapter 5

Labelled Vulnerability Dataset Generation

The challenge of identifying vulnerabilities in Android source code has been exacerbated by the absence of an accurate AI-driven method, during the coding process, primarily due to the lack of adequately labelled training datasets. This research presents the LVDAndro dataset as a solution to this problem. A collection of Android source code vulnerability datasets, categorised according to CWE identifiers (IDs), were compiled. By altering the quantity and sources of apps, three distinct datasets were produced through app scanning. The LVDAndro, containing over 2,000,000 unique code samples, was obtained by scanning more than 15,000 apps. This chapter elaborates on the dataset's creation process and its characteristics.

5.1 Dateset Generation Process

The dataset includes a wide range of source code samples, each exhibiting varying degrees of complexity and security vulnerabilities. The creation process consists of three main stages, as outlined below.

1. Scrapping APKs and their associated source files (Data collection).
2. Scanning APKs for vulnerabilities with the help of existing tools to label the source code with CWE-IDs (Data labelling).
3. Creating the processed dataset (Preprocessing).

5.1.1 Scrapping APKs and Source Files (Data Collection)

The first step in the formation of the LVDAndro dataset involves the extraction of APKs and their corresponding source code from various application repositories. These include Google Play, Fossdroid [218], AndroZoo [215], and several notable malware repositories [16]. A web crawler was developed to download APKs and their source code from GitHub repositories. An experiment was conducted to examine the potential of using source code from reverse-engineered APKs to construct the dataset, as an alternative to relying on the original source code. This was due to the limited availability of open-source APKs and the abundance of closed-source APKs. Closed-source APKs comprised 76% of the total, while open-source APKs accounted for 24%. Figure 5.1 shows the sources of the apps downloaded for the current version of the dataset, which will be expanded in future versions. The scraped apps belonged to various categories, including trending apps on education, entertainment, tools, multimedia, connectivity, and health.

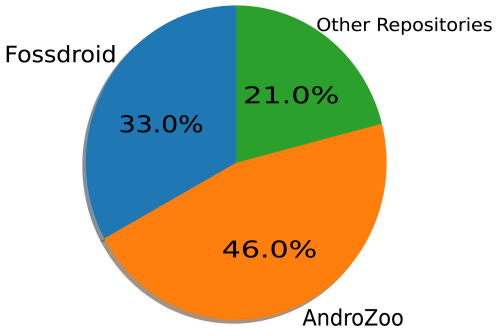


Figure 5.1: Origins of the downloaded applications

5.1.2 Scanning APKs for Vulnerabilities (Data Labelling)

LVDAndro was designed with the intention of employing machine learning for real-time identification of vulnerabilities in source code. A diverse collection of APKs and source files was crucial for the creation of robust and effective machine learning models. During the dataset creation process, a variety of scanning methods were employed to scan both APKs and source files for vulnerabilities. This offered a broad range of vulnerabilities for the training of machine learning models.

The LVDAndro dataset was built using a code analysis approach within the static analysis method. This process involved scrutinising APKs and Android project files, which encompass both the source code and the file structure. Vulnerability scanning tools, such as MobSF and Qark, were employed for this purpose. During the scanning stage,

these tools were capable of identifying the vulnerable lines of code and the associated CWE-IDs. The idea was that the resultant dataset could be leveraged to train machine learning models, allowing the models to learn from the strengths of both scanners and surpass either tool in terms of detection efficiency. A Python script was developed to automate the scanning process, and this script was utilised to scan all the applications.

To analyse an APK or Android project with MobSF, it needs to be set up as a server, enabling the execution of various API requests like upload, scan, and download. Upon uploading an APK or project, MobSF decompiles it using tools such as JADX, dex2-jar, and JD-GUI. The decompiled source code or project files are then scanned for potential vulnerabilities. After the scan is complete, the results are stored in a local database and associated with a generated hash value. These results are retrieved as a JSON object and passed to an automation Python script using the hash value. This JSON object contains details about the uploaded files, including their vulnerability status, manifest analysis details, code analysis details, and related files. A separate Python script is used to extract the necessary details and the source code lines for both vulnerable and non-vulnerable codes as labelled by MobSF.

To perform an analysis using Qark, it needs to be run as a shell script since it does not offer APIs like MobSF. The APK or the directory of the project source file should be provided as parameters when initiating Qark. If an APK is given, Qark decompiles it using tools such as Fernflower, Procyon, and CFR, and then scans it to identify vulnerable lines of code. If a file is provided, it is directly scanned. Once the vulnerable lines of code are detected, a Python script labels and stores them, along with a description of the scanner, the type of vulnerability, and the severity level.

Python scripts were developed to examine APKs and source files, employing a unified approach that merges the functionalities of MobSF and Qark. Based on the analysis, it was found that there were no conflicting classifications between MobSF and Qark. However, the number of vulnerable categories each scanner could identify varied. These scripts utilise techniques to scan and identify potential vulnerabilities in the application or source code. The results are tagged with CWE IDs to provide relevant information. The code, vulnerability status, and category, together with other fields as listed in Table 5.2, were also captured.

5.1.3 Creating Processed Dataset (Preprocessing)

In this stage, several pre-processing steps were implemented. First, user-defined string values were replaced with user_str, as ordinary user-defined string values do not significantly contribute to vulnerabilities [219]. However, string values that include IP addresses and encryption algorithms such as AES, SHA-1, and MD5 were not replaced, as they could lead to vulnerabilities like CWE-200 and CWE-327. These vulnerabilities are related to the exposure of sensitive information to unauthorised parties and the use of insecure cryptographic algorithms. Following that, all comments were replaced with *//user_comment*, considering that language compilers ignore comments. Finally, duplicates were removed based on the processed code and the vulnerability status.

5.2 Resulting Dataset

During the creation of the LVDAndro dataset, multiple datasets were generated. This section explores the characteristics of these resulting datasets.

5.2.1 Different Datasets

The LVDAndro datasets were created by analysing real-world Android apps. Dataset 01 was put together by including all the well-known open-source APKs and their corresponding Android projects from FossDroid, leading to a total of 511 Apps. Another dataset, known as Dataset 02, consisted of 5,503 open-source APKs and their associated projects, scanned from all the apps listed in FossDroid across 17 different categories such as Internet, Systems, Games, and Multimedia. Furthermore, Dataset 03 was formed by scanning 15,021 APKs from FossDroid, AndroVul, and Android malware repositories. This dataset encompasses scanned source code from both open-source and closed-source applications, containing 23 different CWE ID labels. If there is a requirement to build AI models based on the types of apps, the three variations of datasets can be used. However, if there is no such need, Dataset 3 can be used to perform a thorough analysis and build more accurate models as it includes a large number of labelled source code examples. A synopsis of the LVDAndro datasets is given in Table 5.1.

Following the processing of Dataset 01 and Dataset 02, nine sub-datasets were formed. These sub-datasets were created using three scanning methods: the MobSF scanner, the Qark scanner, and the combined scanner, with the objective of assessing their effectiveness. For each method, three sub-datasets were created using exclusively APKs, solely

Table 5.1: Overview of the LVDAndro datasets

Dataset	**No. of Code Samples**	**No. of Vulnerable Codes**	**No. of Non Vulnerable Codes**	**Vul : Non-vul Ratio**	**No. of CWE-IDs**	**Description**
Dataset 01	1,020,134	765,101	255,034	1:3	22	Created with the use of 511 open-source applications. Nine sub-datasets were created - each scanned with MobSF, Qark, and Combined scanners (three were generated by scanning only APKs, three by scanning only source files, and three by scanning both APKs and source files).
Dataset 02	14,228,925	10,529,405	3,699,521	7:9	23	Created with the use of 5,503 open-source apps. Nine sub-datasets were produced - each scanned with MobSF, Qark, and the Combined scanners (three were created by scanning solely APKs, three by scanning only source files, and three by scanning a combination of both APKs and source files).
Dataset 03	21,289,029	14,689,432	6,599,597	9:11	23	Created with the use of 15,021 apps. One dataset was created - it was scanned with the combined scanner using a mix of both open-source and closed-source applications from various sources.

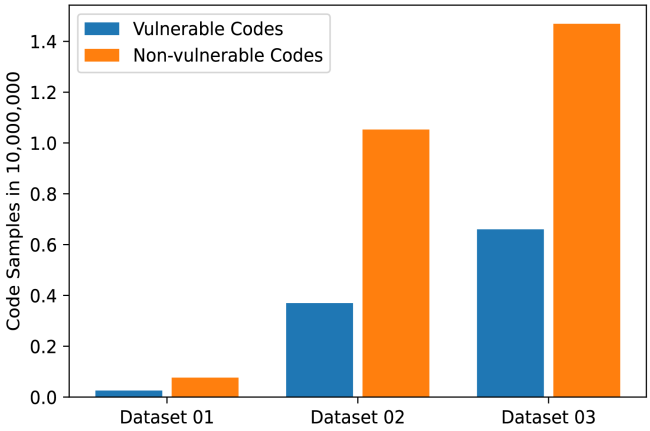


Figure 5.2: Distribution of vulnerable and non-vulnerable code samples in each dataset (using the APK Combined Approach)

Android projects, and a combination of both APKs and Android projects. Dataset 03, which was created using only APKs and the combined method, resulted in one dataset.

Figure 5.2 depicts the spread of code samples, both vulnerable and non-vulnerable, throughout the LVDAndro datasets. A review of the data reveals that the number of non-vulnerable source code samples generally exceeds that of the vulnerable source code samples. As the datasets were derived from actual applications, they might contain a significant portion of non-vulnerable code.

5.2.2 Statistics of Datasets

Java and Kotlin are the predominant programming languages used for native Android application development [220]. Among these, Java has a longer history of extensive use. Consequently, the LVDAndro dataset 03 includes 17,882,784 lines of Java code, making up 84% of the total source code. Table 5.2 presents the fields included in the LVDAndro dataset. While the processed code, vulnerability status, and CWE-IDs are crucial for detecting vulnerabilities, additional fields could provide extra information useful for making predictions. The CWE-IDs that can be found in LVDAndro are categorised based on the likelihood of their exploitation listed in the Table 5.3. The distribution of CWE-IDs in the LVDAndro Dataset 03 is depicted in Figure 5.3.

A significant number of code samples are associated with CWE-532, which is expected given the common practice of logging for debugging purposes. However, these logs may unintentionally include sensitive data input by the developer. Similarly, CWE-312 has

Table 5.2: Fields in LVDAndro

Field Name	**Description**
Index	Auto-generated identifier
Code	The line of source code in its original form
Processed_code	The line of source code following the preprocessing stage
Vulnerability_status	Vulnerable (1) or Non-vulnerable (0)
Category	Category of the vulnerability
Severity	Intensity of the vulnerability
Type	Type of the vulnerability
Pattern	Pattern of the vulnerable code
Description	Description of the vulnerability
CWE_ID	CWE-ID of the vulnerability
CWE_Desc	CWE-based description of the vulnerable class
CVSS	Common vulnerability scoring system
OWSAP_Mobile	Open web application security project for mobile apps details
OWSAP_MASVS	OWASP Mobile application security verification standard
Reference	CWE reference URL for the vulnerability

a large number of code samples due to the frequent occurrence of developers storing sensitive data in plain text. While most other CWE categories have a fairly even distribution of code samples, categories such as CWE-299, CWE-502, and CWE-599 have fewer instances. This is attributed to their complex nature and the difficulty in identifying relevant examples within the Android source code.

Figure 5.4 presents the distribution of CWE-IDs in Dataset 03, considering the likelihood of each CWE being exploited. As the dataset contains 95% of code samples that are susceptible to both high and moderate exploitability CWE-IDs, it is expected to be highly efficient in detecting vulnerabilities.

Table 5.3: Available CWE-IDs in LVDAndro

Likelihood of Exploit	**CWE-ID**	**CWE Description**
High	CWE-79	Improper Neutralisation of Input During Web Page Generation ('Cross-site Scripting')
High	CWE-89	Improper Neutralisation of Special Elements used in an SQL Command ('SQL Injection')
High	CWE-200	Exposure of Sensitive Information to an Unauthorised Actor
High	CWE-295	Improper Certificate Validation
High	CWE-297	Improper Validation of Certificate with Host Mismatch
High	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
High	CWE-330	Use of Insufficiently Random Values
High	CWE-599	Missing Validation of OpenSSL Certificate
High	CWE-649	Reliance on Obfuscation or Encryption of Security-Relevant Inputs without Integrity Checking
High	CWE-676	Use of Potentially Dangerous Function
High	CWE-926	Improper Export of Android Application Components
High	CWE-927	Use of Implicit Intent for Sensitive Communication
High	CWE-939	Improper Authorisation in Handler for Custom URL Scheme
Medium	CWE-250	Execution with Unnecessary Privileges
Medium	CWE-276	Incorrect Default Permissions
Medium	CWE-299	Improper Check for Certificate Revocation
Medium	CWE-312	Cleartext Storage of Sensitive Information
Medium	CWE-502	Deserialisation of Untrusted Data
Medium	CWE-532	Insertion of Sensitive Information into Log File
Medium	CWE-919	Weaknesses in Mobile Applications
Medium	CWE-921	Storage of Sensitive Data in a Mechanism without Access Control
Medium	CWE-925	Improper Verification of Intent by Broadcast Receiver
Low	CWE-749	Exposed Dangerous Method or Function

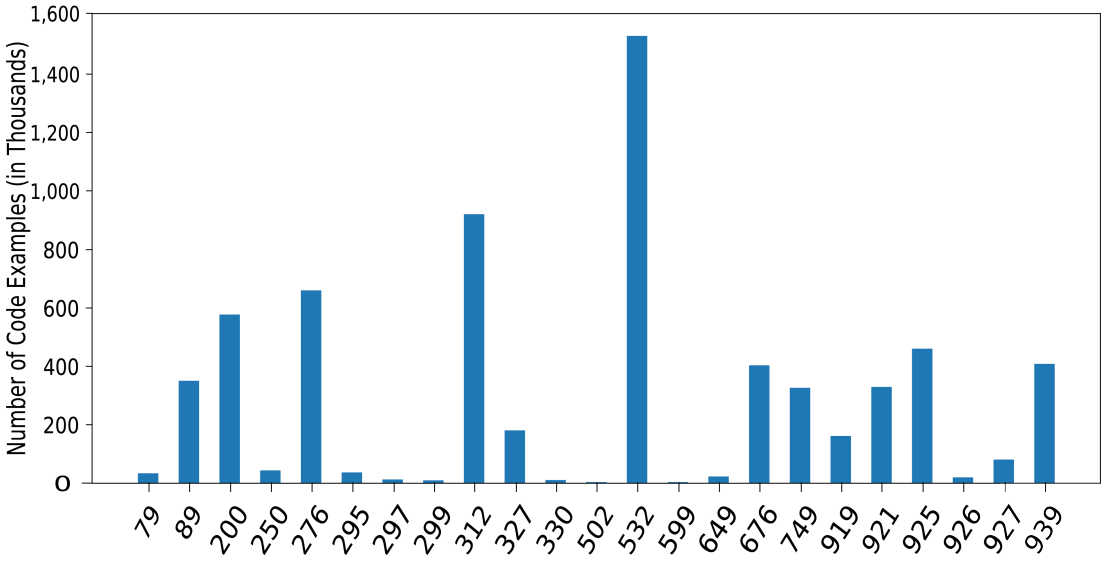


Figure 5.3: Distribution of CWE-IDs in dataset 03

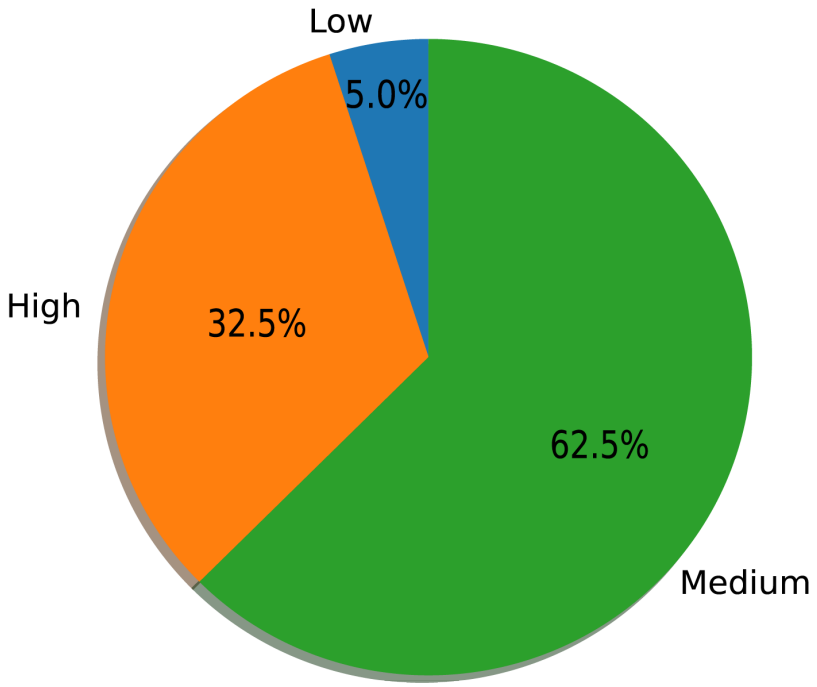


Figure 5.4: Spread of CWE-IDs according to the likelihood of exploitation

5.3 Chapter Summary

This chapter's primary objective is to present the LVDAndro dataset and its features, which can be utilised to create AI models for identifying Android source code vulnerabilities. The LVDAndro dataset, the only verified dataset labelled according to CWE-IDs and encompassing a vast amount of Android source code scanned from actual applications, is a significant contribution to the field. LVDAndro was created using a hybrid scanning approach with multiple scanners and contains over 2 million unique code samples labelled by CWE categories from more than 15,000 Android apps, gathered through a specially built web crawler. The LVDAndro scanner's capabilities have a few limitations due to the scarcity of high-performing open-source Android app scanners. Furthermore, the LVDAndro dataset contains more non-vulnerable source code samples than vulnerable ones due to the nature of the source code in real applications. Despite these limitations, this dataset propels Android app security research forward and assists developers in understanding and mitigating vulnerabilities. Hence, this chapter, with the help of the LVDAndro dataset, addressed RQ2 and accomplished RO2 as specified in chapter 1.

Chapter 6

Vulnerability Detection using AI-based Model

As highlighted in chapter 5, an AI-based approach can be employed to identify vulnerabilities in Android code with high precision, given that a correctly labelled dataset is used. Therefore, this chapter discusses the use of the LVDAndro dataset to construct AI-based models. Initially, the concept's feasibility is confirmed using AutoML, and the most appropriate dataset from the LVDAndro series is chosen. Following that, the potential of an ensemble model for an improved model is discussed, along with the demonstration of the applicability of deep learning techniques to construct the AI-based model. In addition to that, the usage of XAI, which can aid in understanding the model's prediction results, is also discussed.

6.1 Proof-of-Concept Demonstration with AutoML

This section presents the proof-of-concept demonstration using LVDAndro to train machine learning models, to identify vulnerabilities in Android source code. It illustrates that an AutoML model, when trained on the LVDAndro dataset, can effectively recognise and categorise various kinds of vulnerabilities in the Android source code.

6.1.1 Training AutoML Models

This section compares the effectiveness of AutoML models developed based on auto-sklearn [221, 222] that have been trained on LVDAndro datasets for identifying lines of

code that are vulnerable (binary classification) and for identifying CWE-IDs (multi-class classification). To address the issue of data imbalance, the data was resampled with an equal ratio of vulnerable to non-vulnerable samples using the NearMiss undersampling technique and the dataset was divided into a 75:25 proportion for training and testing purposes. The n-gram technique was employed to create the feature vectors for these tasks, with an ngram_range of 1,3 and a minimum document frequency (min_df) of 100 and a maximum document frequency (max_df) of 40. Then the AutoML models were executed, and the performance metrics obtained are displayed in Table 6.1 for binary classification and in Table 6.2 for multi-class classification, and they are organised according to the dataset.

Table 6.1: Performance comparison of AutoML models in binary classification

Sub dataset Name	**Binary Classification**		
	Accuracy	**F1-Score**	**Top Classifier**
Dataset 01			
APKs Qark	91%	0.90	RF
Source Qark	91%	0.90	RF
All Qark	91%	0.90	MLP
APKs MobSF	91%	0.90	RF
Source MobSF	91%	0.90	SVC
All MobSF	91%	0.90	MLP
APKs Combined	92%	0.91	MLP
Source Combined	92%	0.90	MLP
All Combined	92%	0.90	MLP
Dataset 02			
APKs Combined	93%	0.92	RF
Source Combined	93%	0.91	RF
All Combined	93%	0.91	RF
Dataset 03			
APKs Combined	94%	0.94	RF

As per Table 6.1 and Table 6.2, it is evident that the combined method, which uses APKs, source files, and both, yielded superior results for models in Dataset 01. Therefore, only the combined method was employed to train AutoML models in Dataset 02. During the training with Dataset 02, it was found that the APKs combined method surpassed the other source combined and all combined methods. Hence, only APKs were utilised for scanning in Dataset 03. Moreover, downloading APKs from multiple sources could potentially diminish bias and enhance overall performance. An increase in the dataset

Table 6.2: Performance comparison of AutoML models in multi-class classification

Sub dataset Name	Multi-class Classification		
	Accuracy	**F1-Score**	**Top Classifier**
Dataset 01			
APKs Qark	91%	0.82	RF
Source Qark	91%	0.81	RF
All Qark	91%	0.81	RF
APKs MobSF	91%	0.84	RF
Source MobSF	91%	0.83	RF
All MobSF	91%	0.83	RF
APKs Combined	92%	0.88	RF
Source Combined	92%	0.84	RF
All Combined	92%	0.86	RF
Dataset 02			
APKs Combined	93%	0.91	RF
Source Combined	93%	0.85	RF
All Combined	93%	0.87	RF
Dataset 03			
APKs Combined	94%	0.93	MLP

size led to a consistent improvement in F1-Scores for both binary and multi-class classifications. Reducing false positives and false negatives is vital to improve the efficiency of any ML-based solution, with a greater emphasis on reducing false negatives in this problem. To achieve this, several steps, such as enhancing data quality by removing duplicates, handling missing values and manual verification of labels, were performed during preprocessing and training was undertaken to decrease both types of false alarms.

6.1.2 AutoML Model Comparison

An API was constructed to identify lines of code that are vulnerable and the corresponding CWE-ID, utilising an AutoML model that was trained on the LVDAndro dataset. The API takes lines of source code as input, and in this experiment, it was evaluated on a collection of 3,312 lines of source code (unseen data) that included both vulnerable examples from the CWE repository and non-vulnerable examples from actual applications. The vulnerable code sample has a similar distribution to LVDAndro as in Figure 5.3. Following that, an APK was generated that included the same 3,312 lines of source code, and this APK was scanned using MobSF and Qark Scanners. The accuracies of these three methods are documented in Table 6.3.

Table 6.3: Accuracy comparison of proposed ML model with MobSF and Qark

Approach	TP	TN	FP	FN	Precision	Recall	Accuracy	F1-Score
MobSF	609	588	84	31	0.8788	0.9838	91.23%	0.9284
Qark	590	580	88	54	0.8721	0.9306	89.18%	0.8994
Proposed	625	609	59	19	0.9137	0.9952	94.05%	0.9527

The detection methods used by MobSF and Qark are signature-based, which are notorious for generating a large number of false negatives, while still maintaining a high level of accuracy in terms of true positives. To address this issue, the suggested ML technique trained on the LVDAndro dataset can be employed, as it has learned from the strengths of both scanners, making it more robust. With an accuracy rate of 94%, the proposed ML model was able to accurately predict the vulnerability associated with the code being tested. This test was conducted using unseen source code, demonstrating that the proposed method can detect vulnerabilities in new APKs with a high degree of accuracy, thereby validating the hypothesis - proof of concept (PoC). The accuracy of the proposed method can be further enhanced by integrating additional scanners into the pipeline and regularly updating the dataset to include data related to new vulnerabilities. Furthermore, the size and quality of the labelled dataset can also be improved. The LVDAndro GitHub Repository[1] contains the source code and results of these experiments.

6.2 Improving the Capabilities with Ensemble model

The PoC revealed the relevance of the LVDAndro Dataset 03. The need to determine how this dataset can be used for an ensemble model was identified, to create a more generalised and enhanced model. Statistics related to Dataset 03 are presented in Table 6.4.

Table 6.4: Statistics of the LVDAndro Dataset

Characteristic	Value
No. of Used APKs	15,021
No. of Vulnerable Code Lines	6,599,597
No. of Non-Vulnerable Code Lines	14,689,432
No. of Distinct CWE-IDs	23

In datasets derived from real applications, the quantity of non-vulnerable source code samples typically surpasses that of vulnerable samples. This issue of data imbalance

[1]https://github.com/softwaresec-labs/LVDAndro

was tackled during the construction of the AI-based model by down-sampling the non-vulnerable instances in the dataset. Vulnerable code instances comprise lines of code for 23 different CWE-IDs, as depicted in Figure 5.3. However, for certain CWE-IDs, namely CWE-79, CWE-250, CWE-295, CWE-297, CWE-299, CWE-327, CWE-330, CWE-502, CWE-599, CWE-649, CWE-919, CWE-926, and CWE-927, there are only a few examples of vulnerable code. To address this, a new class called *Other* was introduced and utilised to reassign the labels for these source code samples.

During the model's construction, the LVDAndro dataset was split into a 75% portion for training and a 25% portion for testing. Given that the model is required to predict both the vulnerability status and the vulnerability category based on CWE, two classification tasks were carried out: binary and multi-class classification, continuing the previous work in [20]. The n-gram technique was employed to create the feature vectors for these tasks, with an ngram_range of 1,3 and a minimum document frequency (min_df) of 100 and a maximum document frequency (max_df) of 40, resulting in two feature vectors. For the binary classification, the feature vector was formed using the processed_code and vulnerability_status, whereas for the multi-class classification, the feature vector was formed using the processed_code and CWE-ID.

In order to identify the classifiers that are effective in both binary and CWE-based multi-class classification, a range of commonly used learning classifiers, including NB, LR, DT, SVM, RF, GB, XGB, and MLP, were examined [17]. Following this, an ensemble learning model was constructed using the Stacking classifier from Scikit-learn [223], with the previously examined learning classifiers serving as estimators. Stacking is a powerful ensemble learning technique that leverages the strengths of multiple models to improve overall predictive performance. The fundamental concept of stacking is to combine various base models in such a way that they can collectively enhance each other's outputs. This is achieved by training a stacking classifier, also known as a meta-classifier, on the predictions generated by these base models. The meta-classifier is not just an additional layer of prediction but rather a model that learns from the combined predictions of the base models. By doing so, it can identify patterns in the errors and biases of the individual base classifiers and correct them, potentially leading to better overall performance.

In contrast to other ensemble methods like bagging and boosting, which rely on averaging or weighted voting of predictions from multiple models, stacking involves the training of a meta-model. This meta-model learns to optimally integrate the predictions from the various base models in a more nuanced and sophisticated manner. The meta-classifier

has the ability to discern when and how much to trust the predictions of each base model, effectively learning how to combine them in a way that maximises predictive accuracy. This additional layer of learning in stacking can lead to significant performance improvements, especially in complex predictive tasks where no single model excels across all scenarios. The rationale behind employing stacking is that this method capitalises on the diversity of different base models, turning their individual weaknesses into strengths through an intelligently crafted meta-model, ultimately yielding a more robust and accurate final prediction [224]. The ensemble model underwent evaluation using five-fold cross-validation, and the prediction probability, decision function, and predictions were assessed for each estimator. The reason for using five-fold cross-validation rather than other n-fold ($n \neq 5$) is the trade-off between computation time and the reliability of the performance estimate. Five folds are often seen as a good balance,

The effectiveness of each classifier and the proposed ensemble model was evaluated based on their F1 scores and accuracies in binary classification for detecting vulnerability status and multi-class classification for identifying CWE categories. The outcomes of the comparison, which include accuracies and the macro averages of precision, recall, and F1-score, are displayed in Table 6.5. The suggested ensemble model attains high combinations of accuracy, precision, recall, and F1-score in both binary and multi-class classifications. This outcome is likely attributable to the ensemble stacking classifier's ability to amalgamate the strengths of all the other classifiers. The trained ensemble model exhibits an accuracy of 95% for both binary and multi-class classification, with F1-scores of 0.95 and 0.93 for binary and multi-class classification, respectively. The precision, recall, and F1-score values for each CWE-ID in the multi-class classification are detailed in Table 6.6.

Table 6.5: Performance Comparison of Learning Models

Model	**Binary Classification**				**Multi-class Classification**			
	Accuracy	**Precision**	**Recall**	**F1-Score**	**Accuracy**	**Precision**	**Recall**	**F1-Score**
NB	91%	0.91	0.91	0.91	88%	0.86	0.89	0.87
LR	94%	0.94	0.94	0.94	94%	0.92	0.92	0.92
DT	94%	0.94	0.94	0.94	92%	0.90	0.90	0.90
SVM	89%	0.89	0.88	0.88	89%	0.88	0.87	0.88
RF	94%	0.94	0.94	0.94	93%	0.91	0.90	0.90
GB	91%	0.92	0.91	0.91	92%	0.92	0.91	0.91
XGB	94%	0.94	0.94	0.94	93%	0.92	0.92	0.92
MLP	93%	0.93	0.93	0.93	92%	0.91	0.90	0.90
Ensemble Model	95%	0.95	0.95	0.95	95%	0.94	0.93	0.93

Table 6.6: F1-Score for each CWE-ID with Ensemble Model

CWE-ID	Precision	Recall	F1-Score
CWE-89	1.00	1.00	1.00
CWE-200	0.94	0.96	0.95
CWE-276	0.97	0.98	0.97
CWE-312	0.93	0.95	0.94
CWE-532	0.98	0.99	0.99
CWE-676	1.00	1.00	1.00
CWE-749	0.65	0.90	0.76
CWE-921	0.95	0.90	0.93
CWE-925	0.99	0.99	0.99
CWE-939	0.92	0.71	0.80
Other	0.96	0.90	0.93

The ACVED GitHub Repository[2] contains the source code and results of these experiments.

6.3 Development of Neural Network-Based Vanilla Models

The use of models based on neural networks for the same objective is also investigated using the LVDAndro dataset since the customisation and enhancements that can be done to the ensemble model are limited. LVDAndro offers processed data, and the fields *processed_ code* and *vulnerability_ status* were used for binary classification analysis. To ensure a balanced dataset and reduce class bias, the first step was to balance the dataset with an equal ratio of vulnerable to non-vulnerable samples using the NearMiss undersampling technique. Following this, the dataset was divided, with 75% allocated for training and 25% for testing. The feature vector was constructed as similar to the proof-of-concept of LVDAndro, using the n-grams technique, with *ngram_ range* set to 1-3, *min_ df* set to 40, and *max_ df* set to 0.80.

This feature vector was subsequently used to train a neural network model. Several neural network models were tested with different layer configurations, and it was found that the model with one hidden layer containing 20 perceptrons and an output layer with two nodes outperformed the others in terms of performance, accuracy, and F1-Score. The *relu* activation function was used for the input and hidden layers, while the *sigmoid* activation function was applied to the output layer, as it showed good results

[2]https://github.com/softwaresec-labs/ACVED

in experiments. To prevent overfitting, early stopping was implemented with the help of a grid search. This involved monitoring *test_ loss* and setting parameters to *min_ delta* = 0.0001 and *patience* = 20 in *auto* mode. The training process of the neural network model used the *Adam* optimiser with a default *learning rate* of 0.001, and the binary cross-entropy loss function was used.

For multi-class classification, the feature vector was built using the *processed_ code* and *CWE-ID* fields. Labels were transformed using one-hot encoding. While LVDAndro includes code samples for 23 CWE categories, some classes have fewer samples due to their characteristics. As a result, only the top 10 classes were kept, and the remaining classes were relabelled as Other. The dataset was then balanced through resampling, and the feature vector was created using the same *ngram_ range* values (1-3), *min_ df* (40), and *max_ df* (0.80) as in the binary classification model. This feature vector was then employed to train a neural network model with an input layer, one hidden layer with 20 perceptrons, and an output layer with 11 nodes. The *relu* activation function was used for the input and hidden layers, while *softmax* was used for the output layer. To mitigate over-fitting, early stopping, similar to the binary classification model (*monitor* = *test_ loss*, *min_ delta* = 0.0001, *patience* = 20, and *mode* = auto), was applied to this model. During the training of the neural network model, the loss function used was categorical cross-entropy, and the *Adam* optimiser was used with the default *learning rate* of 0.001.

6.3.1 Fine-Tuning and Pruning of Vanilla Model Parameters

A series of experiments were carried out to adjust model parameters, including changing the number of hidden layers and the number of perceptrons, in order to find the best setup. In addition, a thorough grid search and hyper-parameter tuning process were performed to confirm the appropriateness of the parameters mentioned earlier. Once the training phase was finished, an evaluation was done to measure the F1-Scores and accuracies for both binary and multi-class classification.

In addition, after tuning the parameters, pruning techniques were applied to the chosen model. Pruning is the process of removing the least important weight parameters in a neural network, with the goal of improving throughput while preserving model accuracy. Magnitude-based pruning is a simple yet effective method for removing weights while maintaining the same level of accuracy. During the training of the model, zeros are gradually assigned to values, which leads to the gradual elimination of insignificant

weights. The accuracy of the model is dependent on the degree of sparsity, so the sparsity level must be carefully chosen to maintain the same level of accuracy. The TensorFlow model optimisation toolbox [225] was used to implement magnitude-based model pruning. The model was initially trained with all parameters and then pruned to achieve 50% parameter sparsity, starting from 0% sparsity.

6.3.2 Performances of the Vanilla Models

Table 6.7 provides a comparison of F1-Scores, accuracies, and model sizes for both binary and multi-class classification tasks using shallow neural networks (SNN). In terms of binary classification, the standard neural network model is labelled as Vanilla-B, while the multi-class classification model is called Vanilla-M. The pruned models, which are designed for binary and multi-class classifications, are named Vanilla-B-P and Vanilla-M-P, respectively.

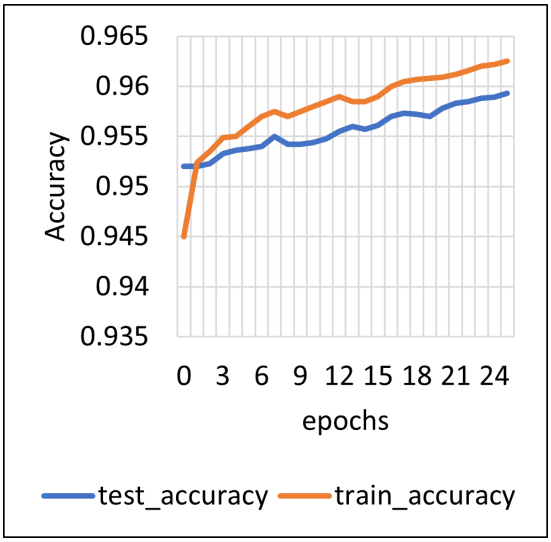
Table 6.7: Performance Comparison of Vanilla Models

Model Name	Accuracy	F1-Score	Model Size
Vanilla-B	96%	0.96	335MB
Vanilla-B-P	95%	0.95	321MB
Vanilla-M	93%	0.91	8.1MB
Vanilla-M-P	92%	0.90	7.9MB

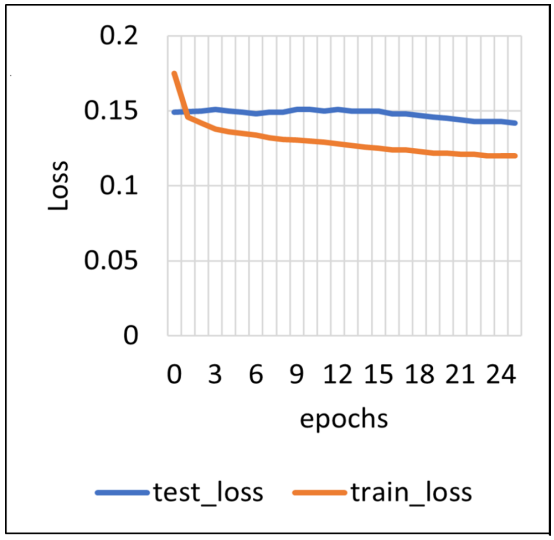
From the data shown in Table 6.7, it can be seen that the unpruned neural network models perform a bit better than the pruned models. This slight difference in performance could possibly be due to the number of example codes utilised and the number of hidden layers used.

For binary classification models, Figure 6.1a shows the changes in training and testing accuracies over epochs, while Figure 6.1b displays the variations in training and testing loss for the same models. In terms of multi-class classification, Figure 6.1c depicts the changes in training and testing accuracies over epochs, and Figure 6.1d presents the profiles of training and testing loss.

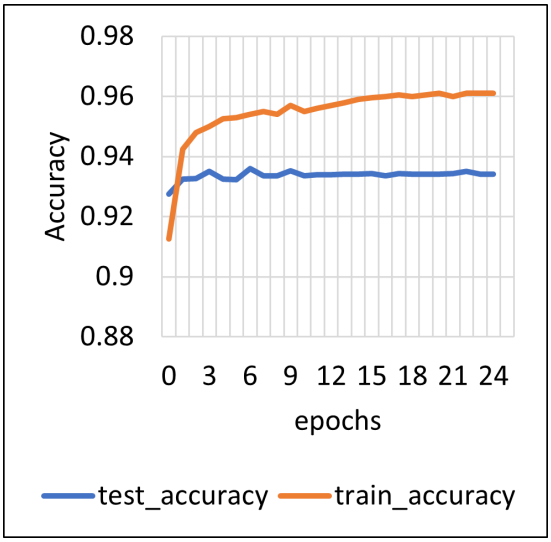
The optimal performance results were obtained with 25 epochs for Vanilla-B and 24 epochs for Vanilla-M. For Vanilla-B, the training accuracy peaked at 96.25%, and the inference accuracy was 95.93%. In this phase, the training loss was 0.12, while the testing loss was 0.142. For Vanilla-M, the best training accuracy of 96.1% and inference accuracy of 93.42% were achieved at the 24th epoch, with corresponding training and testing losses



(a) Accuracy - Vanilla-B



(b) Loss - Vanilla-B



(c) Accuracy - Vanilla-M

(d) Loss - Vanilla-M

Figure 6.1: Accuracy and Loss with Epochs - Vanilla Models

of 0.088 and 0.183. The rise in loss observed during training could indicate that the model is becoming excessively complex and might be fitting noise or outliers in the training data instead of capturing the underlying patterns that apply to new data. Considering that the unpruned models show better performance and the differences in model sizes are negligible, it was decided to choose these unpruned models. When the Vanilla-B model

is compared with the previous Ensemble model's binary classification approach, there is a noticeable 1% enhancement in accuracy in the Vanilla-B model. Despite acknowledging a decrease in model performance, with accuracy falling from 95% to 93% in the multi-class classification model (Vanilla-M) when compared with the previous ensemble method, the neural network models: Vanilla-B and Vanilla-M were retained as the base model for subsequent experiments. This decision was influenced by the potential for model customisations and the introduction of privacy to the model training with federated learning approaches, which will be discussed in the following chapter. The FedREVAN GitHub Repository[3] contains the source code and results of these experiments.

6.4 Incorporating Explainable AI with the AI-based Model and API

For both the binary and multi-class classification models, two pickle files were created. These files hold the trained model, classifier, and vectoriser components. They were then used as inputs in the backend of the newly developed Flask-based web API, which was built using Python. The web API includes a GET request parameter that takes a line of source code from a user and checks it for vulnerabilities. When the web API is initialised, the pre-trained binary and multi-class models are loaded from the pickle files.

Once the vulnerability status and the CWE-ID are predicted, the processed source code is fed into the Lime package in Python, which supports XAI, to get prediction probabilities and explanations for both binary and multi-class models. The Lime package gives the contributions of each word in the processed source code line for both the prediction of vulnerability and the prediction of vulnerable category probabilities. Ultimately, the prediction results are sent back to the user as JSON responses, as illustrated in Figure 6.2a and Figure 6.2b.

The show_in_notebook function from the Python LIME library can offer visual aids for interpreting XAI prediction probabilities. It provides insights into how predictions are made by generating local explanations for individual predictions. LIME works by perturbing the input text and observing the resulting changes in the model's output, thereby identifying the most influential source code words that contribute to the specific classification. Figure 6.3a shows how this can be graphically represented if there are no vulnerabilities in a given line of code. Figure 6.3b shows how Lime-based XAI predictions

[3]https://github.com/softwaresec-labs/FedREVAN

```
{
    "code": "String app_name=\"MyApp\"",
    "processed_code": "String app_name=\"user_str\"",
    "code_vulnerability_status": "Non-Vulnerable Code",
    "code_vulnerability_probability": "0.45635873",
    "probability_breakdown_of_vulnerable_code_words": "",
    "cwe_id": "",
    "predicted_cwe_id_probability": "0",
    "probability_breakdown_of_cwe_related_vulnerable_code_words": "",
    "description": "Non-vulnerable code",
    "mitigation": "Non-vulnerable code",
    "cwe_reference": "Non-vulnerable code"
}
```


(a) Non-vulnerable Code

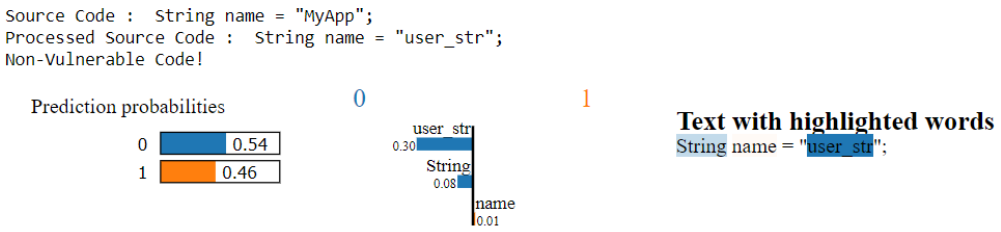

```
{
    "code": "Log.e(\"Login Failure for username :\", \"user123\");",
    "processed_code": "Log.e(\"user_str\", \"user_str\");",
    "code_vulnerability_status": "Vulnerable Code",
    "code_vulnerability_probability": "0.99425936",
    "probability_breakdown_of_vulnerable_code_words": "[('Log', 0.5418730074669474),
    ('user_str', 0.26238023912651687), ('e', 0.10409289309093726)]",
    "cwe_id": "CWE-532",
    "predicted_cwe_id_probability": "0.99",
    "probability_breakdown_of_cwe_related_vulnerable_code_words": "[('Log',
    0.9622543437900251), ('e', -0.0034988003499266235), ('user_str',
    -0.00029704628779474457)]",
    "description": "Information written to log files can be of a sensitive nature and give
    valuable guidance to an attacker or expose sensitive user information.",
    "mitigation": "Try to avoid inserting any confidential information in log statements.
    Minimise using log files in production-level apps.",
    "cwe_reference": "https://cwe.mitre.org/data/definitions/532.html"
}
```


(b) Vulnerable Code

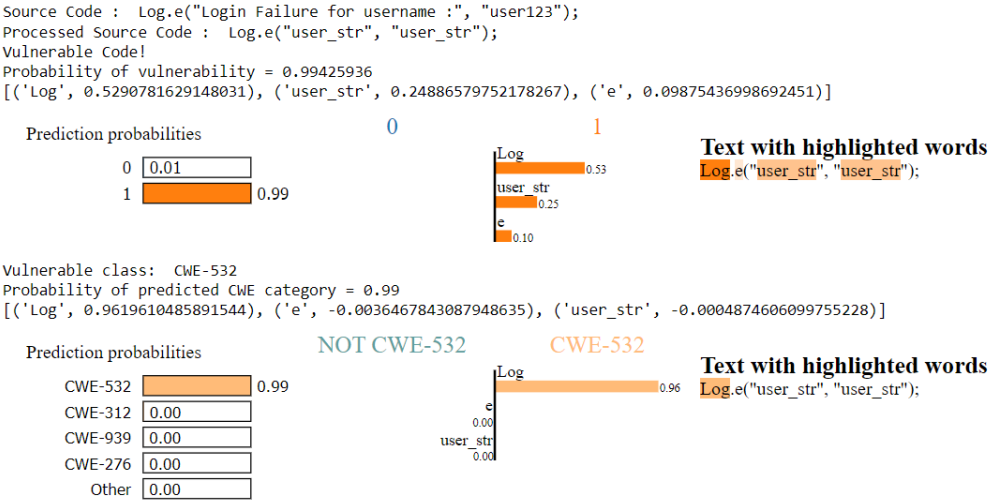
Figure 6.2: Example API Responses for Given Codes

can highlight vulnerable source code, using the example of a line that writes to a log file: *Log.e("Login Failure for username :", "user123");*. This code is linked with CWE-532, which the model accurately predicted with a 0.99 probability. Additionally, the

model identified "Log" as the most significant contributor to the prediction with a 0.53 probability. In multi-class classification, the prediction probability for CWE-532 was 0.99, and the contribution of "Log" to this was 0.96. While this graphical representation will not be utilised for subsequent activities, it can be beneficial for those who wish to comprehend the prediction probabilities and their contributing factors. For the API, details are extracted in a text-based format that mirrors these graphical representations. The descriptions and mitigation suggestions are sourced from the CWE repository [42] to serve as guidance for developers. The developers can review the XAI-based results using the plugin and adjust the code to reduce the probability score associated with the specific code word, while aiming to eliminate the vulnerability entirely. If the developer successfully lowers the probability score of the particular word leading to a vulnerability, the developer is closer to mitigating it. Detailed explanations of how to interpret these results are discussed in chapter 7 when discussing the Android Studio plugin integration process.



(a) Non-vulnerable Code



(b) Vulnerable Code

Figure 6.3: Example XAI Representations for Given Codes

6.5 Chapter Summary

This chapter explores the creation of an AI model that accurately detects vulnerabilities in Android's source code. The model, which integrates ML, AutoML, DL models, and XAI, is built using the LVDAndro dataset. An AutoML model was initially tested using the Auto SKLearn library, showing good performance. However, due to system environment constraints in Auto SKLearn, continuous enhancement was not feasible. Consequently, an ensemble model was experimented with using a stacking classifier, achieving a 95% accuracy rate for both binary and multi-class classifications. Due to the ensemble model's limited customisation capabilities, a deep learning model was experimented with a shallow neural network, resulting in an increase in binary classification accuracy to 96%, but a decrease in multi-class classification accuracy to 93%. This shallow neural network model was further developed with the aim of enhancing detection capabilities through model expansion technologies such as federated learning, as explored in the subsequent chapter. XAI was also incorporated to provide explanations for the predicted vulnerability status and their CWE categories, assisting in their mitigation when using the model. By making a significant contribution in these aspects, this chapter addressed the RQ3 while achieving RO3 and RO4 as specified in Chapter 1.

Chapter 7

Privacy-preserved Community Driven Model Enhancement

Training AI-based models is challenging due to data scarcity. Expanding the LVDAndro dataset is a laborious task as it requires regular scanning of vulnerabilities from actual applications. As data volume increases, so do training time and cost. An alternative could be to source training data from app developers or app development companies. However, privacy concerns arise when acquiring data from these entities, as they may be hesitant to share their proprietary source code. To address this, Federated Learning can be employed to enhance model performance without sharing data, by involving a large number of clients in the training process. While Federated Learning can ensure data security, there is still a risk of data leakage by attackers. To mitigate this risk and enhance privacy, Differential Privacy can be integrated. Then, the entire model and environment can be linked to a blockchain network to create a community-driven model, and perform additional validation to improve model performance. Therefore, this chapter discusses the process of enhancing the current model into a privacy-preserved, community-driven, federated-learning-based model. It also discusses the creation of an Android Studio plugin, along with the procedure for using the plugin in a real-time Android development environment, and an evaluation of the plugin's capabilities.

7.1 Training the Model in a Federated Environment

The basic neural network model, discussed in the preceding chapter, boasts an accuracy of 96% for binary classification (Vanilla-B) and 93% for multi-class classification (Vanilla-M). These models were exclusively trained on the LVDAndro dataset. However, in a practical scenario, the exploration of various datasets, created using a similar methodology to LVDAndro, should be considered. This exploration would involve the integration of multiple training clients with the aim of enhancing the model's performance and capabilities. To achieve this, federated learning is implemented.

Federated learning allows for the gathering of training source code samples from various clients, all while maintaining the privacy of each client's code samples [226]. As such, a simulated federated learning environment was set up, consisting of a server and six clients, as depicted in Figure 7.1. In reality, a large number of clients can be connected with the server. The server, which was run on an Intel Core i5 laptop with 16GB of RAM and Windows 11 OS, was tasked with overseeing the distribution and consolidation of model weights. Four of the clients operated on Ubuntu Linux via Gigabyte Brix (GB-BXBT-2807) devices, while the other two clients, using Intel Core i5 laptops with 8GB of RAM, ran on Windows 10. These clients were in charge of training models with global weights using their individual local datasets.

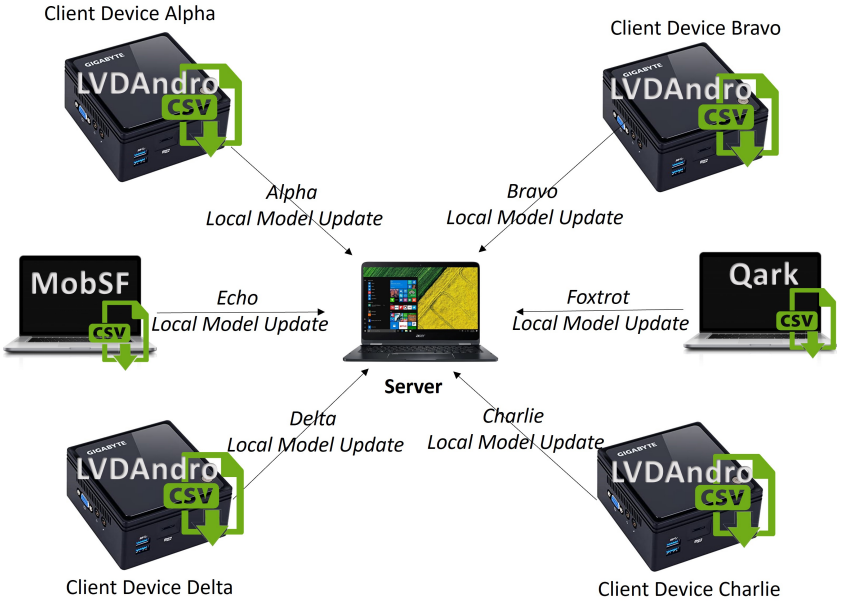


Figure 7.1: Federated Learning Simulated Environment

Both the server and the clients had Python, TensorFlow, and their related dependencies installed. The Flower framework [227] was used, with the server acting as the Flower Server and connecting to the clients. The client named Alpha used the LVDAndro dataset, while three other clients, Bravo, Charlie, and Delta, used the LVDAandro dataset generation mechanism to create datasets from their own data instead of using the LVDAndro dataset. The client Echo used a dataset created by scanning APKs with MobSF, and the client Foxtrot used a dataset created by scanning APKs with Qark. The datasets of Echo and Foxtrot were processed in a manner similar to that used in LVDAndro [19]. The training datasets for each client included the records mentioned in Table 7.1. In real-world scenarios, developers can enhance training by contributing a variety of training data obtained through different methods, such as manual analysis.

Table 7.1: Statistics of the Client Datasets

Characteristic	**Alpha**	**Bravo**	**Charlie**	**Delta**	**Echo**	**Foxtrot**
Used APKs	15,021	5,007	521	622	488	506
Vulnerable Code Lines	6,599,597	1,698,073	389,127	401,196	219,721	188,231
Non-Vul. Code Lines	14,689,432	3,696,846	870,222	881,912	458,211	432,333
Vul. Code Percentage	31.0%	31.5%	30.9%	31.3%	32.4%	30.3%
Distinct CWE-IDs	23	23	22	21	19	18

7.1.1 Performance of the Federated Models

In the federated learning model, the parameters of the neural network model, such as the number of hidden layers, neurons, and optimisers, maintained the same optimal values as those in the Vanilla model. This chosen architecture promotes efficient model convergence and involves a federated communication round of 50, along with five iterations of epochs, as established through the fine-tuning process.

Upon the completion of 50 training rounds, the global model was updated on the federated server and it is now ready for global use. Multiple federated models were developed by varying the clients, in order to study the correlation between the number of participating clients and the performance of the global model in terms of Accuracy and F1-Score. For binary classification models (Federated-B), these models were labelled as Federated-B-a, Federated-B-ab, Federated-B-abc, Federated-B-abcd, Federated-B-abcde, and Federated-B-abcdef. Here, “a” denotes a model trained solely with data from client Alpha, “ab” represents a model trained with data from clients Alpha and Bravo, “abc”

includes data from clients Alpha, Bravo, and Charlie, and so on. A similar naming convention was used for multi-class classification models (Federated-M). The accuracy and F1-Score of the updated models were compared with the Vanilla models, as outlined in Table 7.2a and Table 7.2b.

Table 7.2: Comparison of Federated Models with Vanilla Model

(a) Binary Classification

Model Name	Accuracy	F1-Score
Vanilla-B	96.01%	0.9562
Federated-B-a	96.04%	0.9574
Federated-B-ab	96.07%	0.9596
Federated-B-abc	96.08%	0.9611
Federated-B-abcd	96.11%	0.9641
Federated-B-e	96.03%	0.9546
Federated-B-f	96.02%	0.9551
Federated-B-abcdef	96.17%	0.9649

(b) Multiclass Classification

Model Name	Accuracy	F1-Score
Vanilla-M	93.03%	0.9105
Federated-M-a	93.50%	0.9213
Federated-M-ab	94.02%	0.9311
Federated-M-abc	94.71%	0.9425
Federated-M-abcd	95.08%	0.9503
Federated-M-e	93.31%	0.9209
Federated-M-f	93.19%	0.9201
Federated-M-abcdef	96.02%	0.9624

When the initial Federated binary classification model (Federated-B-a) was compared with the Vanilla binary classification model, there was no significant improvement in accuracy (an increase of 0.03%) or F1-Score (an increase of 0.0012). Interestingly, when all clients were involved (Federated-B-abcdef), the model's performance saw a slight improvement, with an increase in accuracy by 0.16% and F1-Score by 0.0087. The lack of substantial improvement in model performance, when compared to the Vanilla model and other Federated models, could be because Vanilla-B was already well-trained on a large number of samples, thereby reducing the impact of the federated model.

On the other hand, the performance of the multi-class classification models experienced substantial improvements in the federated configuration. As more clients' data was incorporated into the federated setup for multi-class classification, a steady enhancement

in performance was observed. When all clients participated in the federated multi-class classification model (Federated-M-abcdef), there was a significant increase in accuracy by 3.03% compared to the initial model (Vanilla-M), achieving 96.02%. In addition, the F1-Score saw an improvement of 0.0519, reaching 0.9624.

In conclusion, it was clear that incorporating a greater number of clients, especially those using datasets created similarly to LVDAndro, into a federated framework, resulted in significant performance improvements in the overall model that combines binary and multiclass classifications. Moreover, based on this proof of concept, clients can realistically use a variety of scanning methods and actively engage in the training process to contribute to the enhancement of the model.

7.2 Extending to a Blockchain-based Federated Environment

While the federated learning framework can be extended to a wider audience, there is a requirement for a system to verify the model's performance. This ensures that the addition of new client data either enhances or at least sustains the current model's efficiency. However, the lack of incentives for participating clients may hinder its widespread adoption. Simpler approaches, like sharing training data through version control methods or within cloud storage, are impractical due to privacy concerns when involving a larger community in model enhancement using federated learning. To tackle these issues, a blockchain-based method was integrated with the federated model, resulting in a model driven by the community. Using this approach the model can be owned and managed by the community with further enhancement towards removing the central authority. Nevertheless, current blockchain networks such as Ethereum [228] and Hyperledger Fabric [229] possess certain limitations, which obstruct their smooth integration with the proposed specialised model, for real-time detection of Android code vulnerabilities. These limitations encompass issues with scalability, elevated costs, restricted control, absence of reward mechanisms, subscription fees, and restrictions associated with consensus algorithms [230].

As a result, a dedicated private blockchain was developed specifically for the advancement of the Android code vulnerability detection model. Python and Flask were used for its construction. In this setup, the initial block, or the genesis block, was created using a model trained on the LVDAndro dataset. The subsequent blocks are added to

the blockchain network by committed miners who sincerely contribute to enhancing the performance of the model.

Clients must maintain a connection to the Federated server for training purposes. However, they are only permitted to mine a new block if they can satisfy the conditions set by the consensus algorithm, as detailed in algorithm 3. Once these conditions are met, the global model weights are updated, while simultaneously preserving the public ledger with details encrypted in SHA-256 format. Each block is connected to its preceding block by its hash, and it also links to the following block through hash. The updated model weights and the global model are then distributed to the miners who successfully generate and incorporate a new block into the network while updating the global model.

Algorithm 3: Consensus Algorithm

Input: MN: New Model
MC: Current Model
Result: Updates blockchain and global model

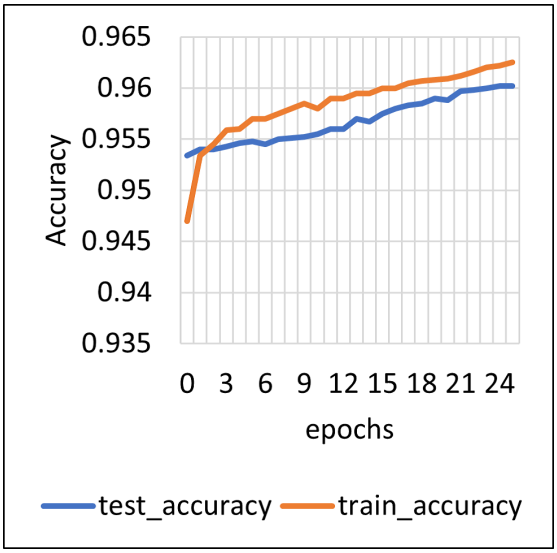
```
if MN_F1-Score ≥ MC_F1-Score and MN_Accuracy ≥ MC_Accuracy then
    Add MN_Weights to the blockchain;
    MC ← MN;
end
```

Considering that the validation process is dependent on the consensus algorithm used, no losses in model performance were expected as it always validates whether the new model performs better than the current model in terms of accuracy and F1-Score. Consequently, the model's accuracy of 96% and F1-Score of 0.96 are consistently anticipated to either remain constant or potentially improve compared to the Federated-B-abcdef and Federated-M-abcdef models. This updated model, then referred to as "Defendroid", includes both binary and multiclass classification models trained within the blockchain-based federated network. The Defendroid GitHub Repository[1] contains the source code and results of these experiments.

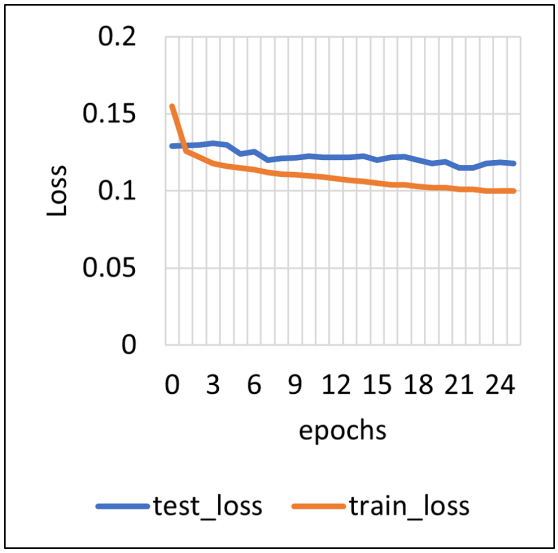
Figure 7.2a illustrates the progression of training and testing accuracies over epochs for Defendroid binary models. In contrast, Figure 7.2b depicts the fluctuations in training and testing loss. Figure 7.2c demonstrates the changes in training and testing accuracies throughout epochs, while Figure 7.2d presents the trends in training and testing loss for the multi-class model.

The optimal performance results were noted at 25 epochs for Defendroid-B and 24 epochs

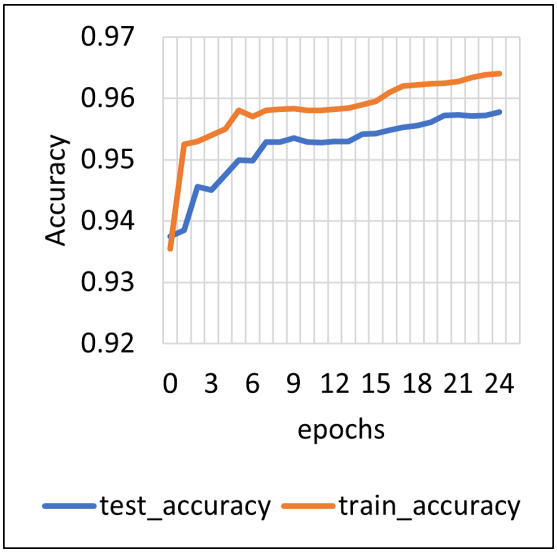
[1] https://github.com/softwaresec-labs/Defendroid



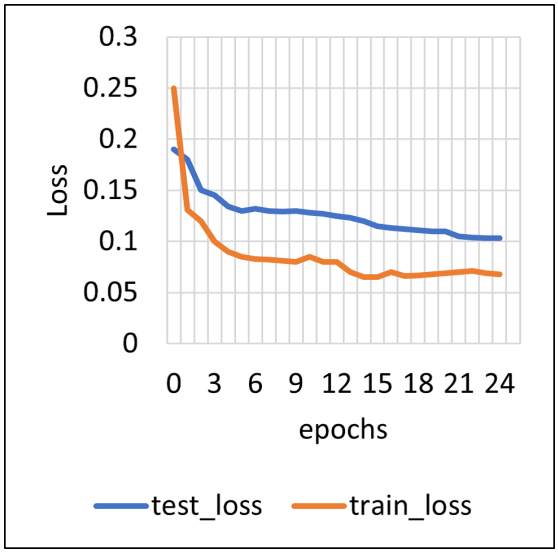
(a) Accuracy - Defendroid-B



(b) Loss - Defendroid-B



(c) Accuracy - Defendroid-M



(d) Loss - Defendroid-M

Figure 7.2: Accuracy and Loss with Epochs - Defendroid Models

for Defendroid-M. For Defendroid-B, the training accuracy peaked at 96.25%, and the inference accuracy was 96.02%. At this stage, the training loss was 0.1, while the testing loss was 0.118. For Defendroid-M, the highest training accuracy reached was 96.4%, and the inference accuracy was 95.78%, both at the 24-epoch point. The corresponding

training and testing losses were 0.068 and 0.103, respectively. These experiments indicated that Defendroid models show a comparatively lower level of over-fitting compared to vanilla models, indicating a substantial enhancement in performance. This could be attributed to the increased number of code samples used for training in the federated learning setup.

7.3 Integrating Differential Privacy

To enhance the confidentiality of the model training phase, techniques of differential privacy were employed. The integration of differential privacy allows for the quantification of privacy assurances. AI models can be responsibly crafted to learn from private data, thereby reducing the potential danger of revealing sensitive training data. After a model is trained using differential privacy, it remains unaffected by any individual training example or a small group of training examples in its dataset. This aids in reducing the risk of sensitive training data exposure in model training. This was achieved through the use of the Google differential privacy framework [231, 232] and the TensorFlow privacy framework [59].

The first step in this process was to execute a grid search. This was done to ascertain the most suitable values for parameters such as *learning rate, noise multiplier, l2_ norm_ clip*, and *num_ microbatches* within differential privacy. The *l2_ norm_ clip* is a hyperparameter that sets the maximum Euclidean (L2) norm for each gradient applied to update model parameters, limiting the optimiser's sensitivity to individual training points. The *noise_ multiplier* refers to the quantity of noise sampled and added to gradients during training. Increased noise generally enhances privacy, potentially at the cost of lower utility. Data batches are divided into smaller units known as microbatches, each ideally containing a single training example. This allows gradients to be clipped on a per-example basis, reducing the negative impact of clipping on the gradient signal and typically maximising utility. The size of microbatches can be increased to include more than one training example to reduce computational overhead. The average gradient across these multiple training examples is then clipped. The *learning_ rate*, a hyperparameter already present in neural network model, influences the impact of each update. A lower learning rate may aid convergence if the updates are noisy [59].

Under perfect conditions, not all participants in federated learning might have privacy as their main concern, with some potentially being more interested in contributing to open-source projects. As a result, it might not be essential to incorporate differential privacy

for every client. To investigate this possibility, a series of experiments were conducted where the number of clients using differential privacy was varied. A grid search was used to find the optimal values for this scenario, which were determined to be a *noise multiplier of 1.3, l2_ norm_ clip of 1.5, batch size of 256, and num_ microbatches of 256.*

7.3.1 Applying Differential Privacy to all Clients

As the first experiment, differential privacy was applied to all clients. As a result, the models were named Defendroid-B-DP-a, Defendroid-B-DP-ab, Defendroid-B-DP-abcd, and so on, where "DP" signifies the use of differential privacy. This same experimental process was also used for the multi-class models, which were named Defendroid-M-DP-a, and so on. The results for the binary models can be found in Table 7.3a, while the outcomes for the multi-class models are shown in Table 7.3b.

Table 7.3: Federated Learning with Differential Privacy Applied to All Clients

(a) Binary classification

Model Name	Accuracy	F1-Score
Defendroid-B-DP-a	95.64%	0.9482
Defendroid-B-DP-ab	95.81%	0.9492
Defendroid-B-DP-abc	95.86%	0.9496
Defendroid-B-DP-abcd	95.98%	0.9504
Defendroid-B-DP-e	95.46%	0.9483
Defendroid-B-DP-f	95.51%	0.9502
Defendroid-B-DP-abcdef	95.65%	0.9501

(b) Multi-class classification

Model Name	Accuracy	F1-Score
Defendroid-M-DP-a	93.01%	0.9201
Defendroid-M-DP-ab	93.17%	0.9212
Defendroid-M-DP-abc	93.33%	0.9242
Defendroid-M-DP-abcd	93.45%	0.9315
Defendroid-M-DP-e	92.11%	0.9156
Defendroid-M-DP-f	92.31%	0.9167
Defendroid-M-DP-abcdef	94.51%	0.9411

7.3.2 Applying Differential Privacy to Randomly Selected Clients

In the second experiment, differential privacy was applied to Alpha, Delta, Echo, and Foxtrot clients. Aside from client Alpha, which utilises the entire LVDAndro dataset, the other clients are selected at random. Therefore, the number and type of clients can

vary in subsequent experiments. This experiment was inspired by real-world situations where privacy concerns are a priority for some clients, but not for others. As a result, the models were named as Defendroid-B-DP-a, Defendroid-B-DP-a-N-b, Defendroid-B-DP-a-N-bc, Defendroid-B-DP-adef-N-bc, and so on. In these labels, "DP" denotes the application of differential privacy, while "N" indicates clients that did not use differential privacy. The same experimental procedure was used for the multi-class models, which were labelled as Defendroid-M-DP-a and so on. The outcomes for binary classification are provided in Table 7.4a, while the results for multi-class classification are outlined in Table 7.4b.

Table 7.4: Federated Learning with Differential Privacy Applied to Randomly Selected Clients

(a) Binary classification

Model Name	Accuracy	F1-Score
Defendroid-B-DP-a	95.64%	0.9482
Defendroid-B-DP-a-N-b	96.01%	0.9508
Defendroid-B-DP-a-N-bc	96.05%	0.9523
Defendroid-B-DP-ad-N-bc	96.06%	0.9532
Defendroid-B-DP-e	95.46%	0.9483
Defendroid-B-DP-f	95.51%	0.9502
Defendroid-B-DP-adef-N-bc	95.81%	0.9541

(b) Multi-class classification

Model Name	Accuracy	F1-Score
Defendroid-M-DP-a	93.01%	0.9201
Defendroid-M-DP-a-N-b	93.4%	0.9261
Defendroid-M-DP-a-N-bc	94.00%	0.9359
Defendroid-M-DP-ad-N-bc	94.20%	0.9410
Defendroid-M-DP-e	92.11%	0.9156
Defendroid-M-DP-f	92.31%	0.9167
Defendroid-M-DP-adef-N-bc	94.78%	0.9452

7.3.3 Applying Differential Privacy to All Clients Except Alpha

In the third experiment, differential privacy was applied to all clients with the exception of Alpha. The reason for this exclusion is that Alpha uses the entire LVDAndro dataset, which is publicly accessible. As a result, privacy considerations are not relevant for this client, while they are assumed for all other clients. In this context, the

models were named as Defendroid-B-N-a, Defendroid-B-DP-b-N-a, Defendroid-B-DP-bc-N-a, Defendroid-B-DP-a-N-bcdef, and so on. Here also, "DP" stands for the application of differential privacy, and "N" denotes clients that did not use differential privacy. The same experimental procedure was used for the multi-class models, which were named Defendroid-M-DP-a and so on. The results for binary classification are provided in Table 7.5a, while the findings for multi-class classification are outlined in Table 7.5b.

Table 7.5: Federated Learning with Differential Privacy Applied to All Clients Except Alpha

(a) Binary classification

Model Name	Accuracy	F1-Score
Defendroid-B-N-a	96.04%	0.9574
Defendroid-B-DP-b-N-a	96.05%	0.9575
Defendroid-B-DP-bc-N-a	96.06%	0.9577
Defendroid-B-DP-bcd-N-a	96.07%	0.9578
Defendroid-B-DP-e	95.46%	0.9483
Defendroid-B-DP-f	95.51%	0.9502
Defendroid-B-DP-bcdef-N-a	96.08%	0.9580

(b) Multi-class classification

Model Name	Accuracy	F1-Score
Defendroid-M-N-a	93.50%	0.9213
Defendroid-M-DP-b-N-a	94.1%	0.9281
Defendroid-M-DP-bc-N-a	94.5%	0.9397
Defendroid-M-DP-bcd-N-a	94.67%	0.9410
Defendroid-M-DP-e	92.11%	0.9156
Defendroid-M-DP-f	92.31%	0.9167
Defendroid-M-DP-bcdef-N-a	95.33%	0.9502

The outcomes of these three experiments underscored the importance of balancing accuracy and privacy in model training. The third experiment, which simulated real-world conditions by allowing client Alpha to use the LVDAndro dataset while other clients used their own data, demonstrated the efficacy of differential privacy measures in maintaining privacy. Furthermore, the privacy budget [232], denoted by epsilon (ϵ), was computed for this setup, resulting in values of 1.19 for binary classification and 1.26 for multi-class classification. These figures suggest a robust privacy guarantee, as a privacy budget within the range of 1 to 1.5 is deemed optimal [233].

7.3.4 Variation of Accuracy and Privacy Budget with Noise Multiplier

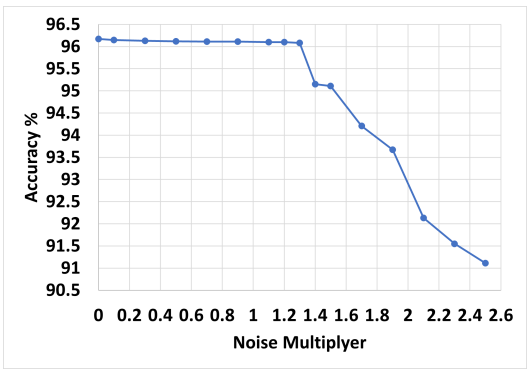
A study was carried out to determine the impact of noise multiplier variations on the performance of the model. For this experiment, differential privacy was applied to all the clients except Alpha. This study entailed altering the noise multiplier within a range of 0 to 2.5 as this range is generally considered to be optimal [233]. It compares the privacy budget (ϵ) and the accuracy for both binary and multi-class classification. Throughout this experiment, the l2_norm_clip, which is associated with gradient clipping for privacy preservation, was kept constant at 0. This calculation was carried out using the Google Differential Privacy framework [232].

The outcomes of the binary classification are detailed in Table 7.6, while Figure 7.3 illustrates this variation. The findings of the multi-class classification are outlined in Table 7.7, while Figure 7.4 provides a visual representation of it.

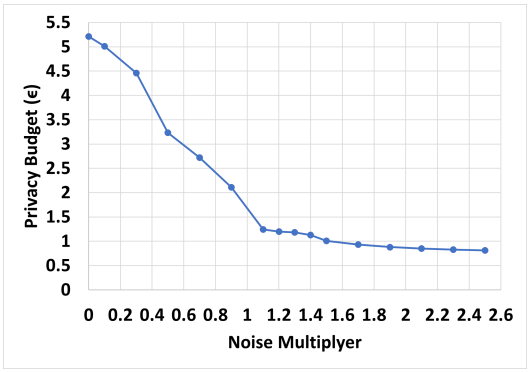
Table 7.6: Variation of Accuracy and Privacy Budget with Noise Multiplier - Binary Classification

Noise Multiplier	**Accuracy**	**Privacy Budget (ϵ)**
0	96.17%	5.21
0.1	96.15%	5.01
0.3	96.13%	4.46
0.5	96.12%	3.23
0.7	96.11%	2.72
0.9	96.11%	2.11
1.1	96.10%	1.24
1.2	96.10%	1.2
1.3	96.08%	1.18
1.4	95.15%	1.13
1.5	95.11%	1.01
1.7	94.21%	0.93
1.9	93.67%	0.88
2.1	92.13%	0.85
2.3	91.55%	0.83
2.5	91.11%	0.81

The findings indicated that a lower noise multiplier corresponds to higher accuracy but a lower privacy budget (indicated by a high ϵ value). Conversely, a high noise multiplier results in a high privacy budget (signified by a low ϵ value) but compromises accuracy. Therefore, to ensure privacy without sacrificing too much accuracy, the noise multiplier should be adjusted to a balanced value.



(a) Accuracy with Noise



(b) Privacy Budget with Noise

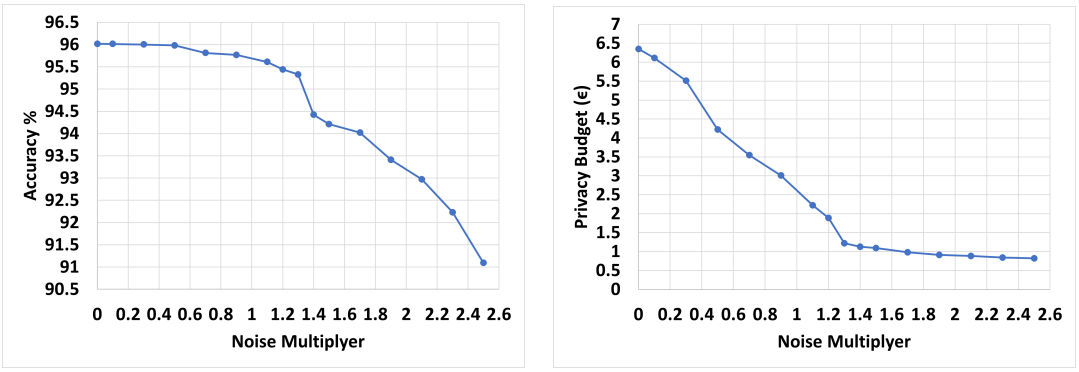
Figure 7.3: Variation of Accuracy and Privacy Budget with Noise Multiplier - Binary Classification

Table 7.7: Variation of Accuracy and Privacy Budget with Noise Multiplier - Multi-class Classification

Noise Multiplier	**Accuracy**	**Privacy Budget (ϵ)**
0	96.02%	6.35
0.1	96.01%	6.11
0.3	96.00%	5.51
0.5	95.98%	4.22
0.7	95.81%	3.55
0.9	95.77%	3.01
1.1	95.61%	2.22
1.2	95.44%	1.89
1.3	95.33%	1.22
1.4	94.42%	1.13
1.5	94.21%	1.09
1.7	94.02%	0.98
1.9	93.41%	0.91
2.1	92.97%	0.88
2.3	92.23%	0.84
2.5	91.09%	0.82

7.3.5 Variation of Accuracy and Privacy Budget with L2_Norm_Clip

A similar experiment was carried out to examine the relationship between accuracy and the privacy budget (ϵ) with respect to l2_norm_clip. Throughout these experiments, the noise multiplier was maintained at 0.



(a) Accuracy with Noise

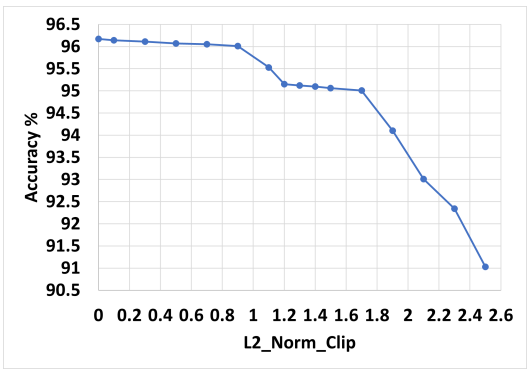
(b) Privacy Budget with Noise

Figure 7.4: Variation of Accuracy and Privacy Budget with Noise Multiplier - Multi-class Classification

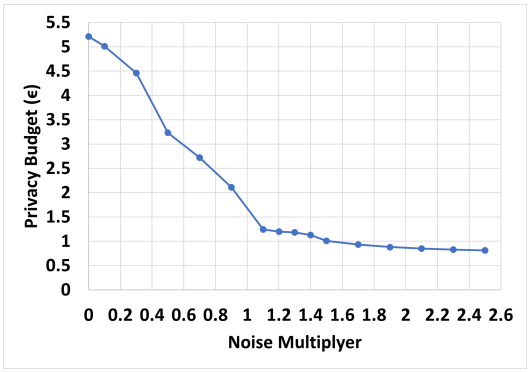
The outcomes of the binary classification are detailed in Table 7.8, and Figure 7.5 provides a visual representation of this variation. The outcomes of the multi-class classification are detailed in Table 7.9, and Figure 7.6 provides a visual representation of this variation.

Table 7.8: Variation of Accuracy and Privacy Budget with L2_Norm_Clip - Binary Classification

L2_Norm_Clip	**Accuracy**	**Privacy Budget (ϵ)**
0	96.17%	5.21
0.1	96.14%	4.98
0.3	96.11%	4.51
0.5	96.07%	3.69
0.7	96.05%	2.96
0.9	96.01%	2.54
1.1	95.53%	1.86
1.2	95.15%	1.65
1.3	95.12%	1.53
1.4	95.10%	1.22
1.5	95.06%	1.02
1.7	95.01%	0.86
1.9	94.10%	0.84
2.1	93.01%	0.81
2.3	92.34%	0.77
2.5	91.03%	0.73



(a) Accuracy with L2_Norm_Clip



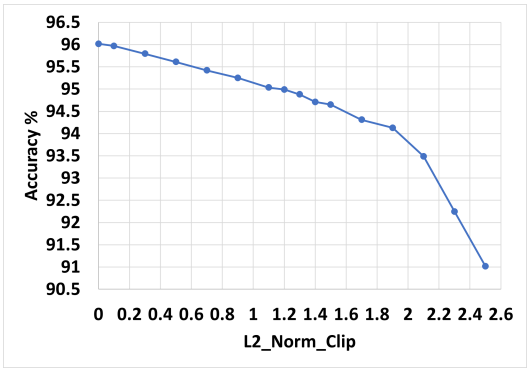
(b) Privacy Budget with L2_Norm_Clip

Figure 7.5: Variation of Accuracy and Privacy Budget with L2_Norm_Clip - Binary Classification

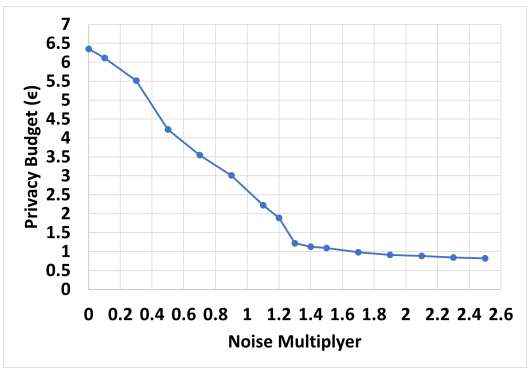
Table 7.9: Variation of Accuracy and Privacy Budget with L2_Norm_Clip - Multi-class Classification

L2_Norm_Clip	**Accuracy**	**Privacy Budget (ϵ)**
0	96.02%	6.35
0.1	95.97%	6.01
0.3	95.79%	5.42
0.5	95.61%	5.03
0.7	95.42%	4.21
0.9	95.25%	3.76
1.1	95.03%	3.06
1.2	94.99%	2.11
1.3	94.88%	1.49
1.4	94.71%	1.21
1.5	94.65%	1.02
1.7	94.31%	0.83
1.9	94.13%	0.74
2.1	93.49%	0.62
2.3	92.25%	0.56
2.5	91.02%	0.48

The analysis of the variations in accuracy and privacy budget, with respect to noise multiplier, and l2_norm_clip, reveals that a compromise in the performance of the model is required to improve privacy. However, finding a balance between them is possible, as elaborated in these experiments.



(a) Accuracy with L2_Norm_Clip



(b) Privacy Budget with L2_Norm_Clip

Figure 7.6: Variation of Accuracy and Privacy Budget with L2_Norm_Clip - Multi-class Classification

7.3.6 Differential Privacy Incorporating Variable Noise and L2_Norm_Clip with Blockchain-based Federated Learning

The ideal values for the noise multiplier and l2_norm_clip are contingent on the training data. Rather than maintaining these values as constants for all training datasets, they should be dynamically adjusted. However, this adjustment should be carried out in a way that does not compromise the performance of the models.

Therefore, a novel algorithm that takes all these factors into account was introduced. This algorithm is built upon the FedAvg algorithm and has the capability to dynamically adjust the noise multiplier and l2_norm_clip for clients who prioritise the privacy of their training data. With the use of this algorithm, it is possible to enhance or maintain the model accuracy and F1-Score while ensuring a high level of privacy, supported by the initially proposed blockchain consensus algorithm. This newly proposed algorithm has been named the *Blockchain-based Federated Averaging with Differential Privacy incorporating Variable Noise and L2_Norm_Clip (DPV-BLFedAvg) Algorithm*, as referenced in algorithm 4.

The DPV-BLFedAvg algorithm operates through a specified number of communication rounds in the federated learning setup and across all clients. For each client, the algorithm first determines if the client has privacy concerns. If so, optimal values for the noise multiplier and l2_norm_clip are chosen. These values are automatically selected via nested loops, iterating from 2.5 to 0 for both the noise multiplier and l2_norm_clip, with each iteration round decreasing by 0.1. The DPAdamOptimiser is employed for

Algorithm 4: Blockchain-based Federated Averaging with Differential Privacy incorporating Variable Noise and L2_Norm_Clip (DPV-BLFedAvg) Algorithm

Input:
N: Total number of clients
K: Number of communication rounds
w_0: Initial global model
α: Learning rate
MN: New Model
MC: Current Model
DP: Differential Privacy Status
ϵ: Privacy Budget
Result: Updated global model w_K
Data:
$max\_nmp = 2.5$: Max Noise Multiplier value
$max\_l2nc = 2.5$: Max L2_Norm_Clip value

```
loop1: for k = 1 to K do
    loop2: for i = 1 to N do
        if DP == TRUE then
            loop3: for x = max_nmp to x = 0 decrement x by 0.1 do
                loop4: for y = max_l2nc to y = 0 decrement y by 0.1 do
                    opt = DPAdamOptimizer(x, y)
                    Train a local model w_{i,k} using client i's local data:
                    w_{i,k} = LocalTraining(w_k, α, opt)
                    Aggregate local model updates:
                    w_agg = (1/N) Σ_{i=1}^{N} w_{i,k} calculate ε;
                    Check the new model's Accuracy, F1-Score and Privacy:
                    if MN_{F1-Score} ≥ MC_{F1-Score} and MN_{Accuracy} ≥ MC_{Accuracy} and
                     1 ≤ ε ≤ 1.5 then
                        Add MN_Weights to the blockchain;
                        MC ← MN;
                        Update the global model:
                        w_{k+1} = w_agg
                        break iterations of loop3 and loop4;
                    end
                end
            end
        else
            opt = AdamOptimizer()
            Train a local model w_{i,k} using client i's local data:
            w_{i,k} = LocalTraining(w_k, α, opt)
            Aggregate local model updates:
            w_agg = (1/N) Σ_{i=1}^{N} w_{i,k} IfMN_F1-Score ≥ MC_F1-Score and MN_Accuracy ≥ MC_Accuracy
             Add MN_Weights to the blockchain;
            MC ← MN;
            Update the global model:
            w_{k+1} = w_agg
        end
    end
end
```

model training. If the new model's accuracy and F1-Score surpass those of the previous model and the privacy budget falls within the range of 1 to 1.5, the new model is updated in the blockchain, and the model weights are updated in the global model. If the client does not have privacy concerns, the accuracy and the F1-Score are the only factors compared after training the local model with AdamOptimiser.

This algorithm consistently preserves or enhances the existing level of accuracy and the F1-Score. If any client attempts to undermine the performance of the model, that model will be rejected and not incorporated into the global model. As a result, an accuracy of 96% and an F1-Score of 0.96 for both binary and multi-class classification are sustained, unless they can be improved further. Moreover, the privacy of the model is assured while achieving a privacy budget range of 1 to 1.5, which is deemed optimal. The final model uses this algorithm.

To validate the algorithm, a series of experiments was carried out by incorporating six additional machines into the existing simulation architecture (Alpha, Bravo, Charlie, etc.). These new machines, designated as N1, N2, N3, N4, N5, and N6, each have a vulnerability dataset labelled using CWE-IDs. Some of these machines employ the LVDAndro approach for dataset generation, while others use source code that includes both vulnerable and non-vulnerable lines of code, labelled based on CWE IDs. The distributions of vulnerable and non-vulnerable code for these new clients can be found in Table 7.10.

Table 7.10: Statistics of New Clients' Data

Characteristic	**N1**	**N2**	**N3**	**N4**	**N5**	**N6**
Vulnerable Code Lines	38,281	123,323	12,224	44,229	35,128	136,967
Non-Vul. Code Lines	97,124	445,235	66,245	125,941	66,324	236,932
Vul. Code Percentage	28.3%	21.7%	15.6%	26.0%	34.6%	36.7%
Distinct CWE-IDs	12	21	15	17	14	22

In these simulated experiments, the previous six clients (Alpha, Bravo, Charlie, etc.) did not have differential privacy applied. The new clients were tested under conditions both with and without differential privacy. In this context, the clients Alpha, Bravo, Charlie, Delta, Echo, and Foxtrot without differential privacy are referred to as *Old*. *DP-N1* refers to the N1 clients with differential privacy enabled, and *N-N2* refers to those without differential privacy. *B* denotes binary classification, and *M* represents multi-class classification. The results for binary classification can be found in Table 7.11, and those for multi-class classification are in Table 7.12.

Table 7.11: Evaluation of DPV-BLFedAvg Algorithm Results - Binary Classification

Model Name	**Noise**	**L2_Norm_Clip**	**Accuracy**	**F1-Score**	**Privacy Budget (ϵ)**	**Model Status**
Defendroid-B-Old	-	-	96.17%	0.9649	-	Accepted
Defendroid-B-Old-N-N1	-	-	96.17%	0.9650	-	Accepted
Defendroid-B-Old-DP-N1	0.5	0.4	96.17%	0.9649	1.2	Accepted
Defendroid-B-Old-N-N2	-	-	96.18%	0.9651	-	Accepted
Defendroid-B-Old-DP-N2	1.0	1.1	96.17%	0.9650	-	Accepted
Defendroid-B-Old-N-N3	-	-	96.17%	0.9623	-	Rejected
Defendroid-B-Old-DP-N3	0	0	96.17%	0.9623	5.6	Rejected
Defendroid-B-Old-N-N4	-	-	96.17%	0.9648	-	Rejected
Defendroid-B-Old-DP-N4	0	0	96.17%	0.9648	4.3	Rejected
Defendroid-B-Old-N-N5	-	-	96.17%	0.9649	-	Accepted
Defendroid-B-Old-DP-N5	0	0	96.17%	0.9649	3.8	Rejected
Defendroid-B-Old-N-N6	-	-	96.18%	0.9652	-	Accepted
Defendroid-B-Old-DP-N6	1.3	1.5	96.17%	0.9650	1.3	Accepted

Table 7.12: Evaluation of DPV-BLFedAvg Algorithm Results - Multi-class Classification

Model Name	**Noise**	**L2_Norm_Clip**	**Accuracy**	**F1-Score**	**Privacy Budget (ϵ)**	**Model Status**
Defendroid-M-Old	-	-	96.02%	0.9624	-	Accepted
Defendroid-M-Old-N-N1	-	-	95.99%	0.9601	-	Rejected
Defendroid-M-Old-DP-N1	1.5	2.3	95.51%	0.9588	1.4	Rejected
Defendroid-M-Old-N-N2	-	-	96.03%	0.9631	-	Accepted
Defendroid-M-Old-DP-N2	1.9	1.8	96.03%	0.9625	1.3	Accepted
Defendroid-M-Old-N-N3	-	-	95.88%	0.9593	-	Rejected
Defendroid-M-Old-DP-N3	0	0	95.88%	0.9593	3.1	Rejected
Defendroid-M-Old-N-N4	-	-	96.02%	0.9625	-	Accepted
Defendroid-M-Old-DP-N4	0	0	96.02%	0.9625	2.2	Rejected
Defendroid-M-Old-N-N5	-	-	95.79%	0.9391	-	Rejected
Defendroid-M-Old-DP-N5	0	0	95.79%	0.9391	1.7	Rejected
Defendroid-M-Old-N-N6	-	-	96.03%	0.9626	-	Accepted
Defendroid-M-Old-DP-N6	1.1	1.3	96.03%	0.9624	1.4	Accepted

The experiments revealed that the base performance of the model consistently maintained a minimum accuracy of 96.17%, an F1-Score of 0.9640 for binary classification, and an accuracy of 96.02%, with an F1-Score of 0.9624 for multi-class classification. When differential privacy was applied, an additional check was conducted to measure the privacy budget, which should also fall within the range of 1 to 1.5. If either of these conditions including minimum accuracy, minimum F1-Score, or privacy budget range was not met, the new model would be rejected and the global model would not be updated. However, the application of differential privacy will increase the training time, as the algorithm requires additional iterations to identify the optimal combination of noise and the l2_norm_clip to achieve the desired privacy budget.

7.4 Application of AI-based Model using an Android Studio Plugin

The revised model acts as the API's backbone, as suggested in chapter 6. By leveraging the model API on the backend, supplemented by XAI, developers are equipped to quickly identify potential code vulnerabilities during active coding. This is achieved by sending the code via the API, using a plugin that is smoothly integrated into their development environment. As a result, developers can effectively examine code for vulnerabilities without having to toggle between various applications. This allows them to rapidly identify and rectify issues as they arise, promoting a seamless workflow without disruptions. This method greatly boosts efficiency, saving both time and crucial resources. According to the review in preceding chapters, currently, there is no plugin available in Android Studio that utilises an AI backend for real-time code vulnerability detection. Hence this is useful for Android app developers to develop secure apps. Comprehensive instructions of integrating and using the plugin available in the GitHub repositories: ACVED[2], FedREVAN[3] and Defendroid[4].

When a user's request is received through the plugin to the API, the procedure begins by using the binary classification vectoriser to convert the line of code. The binary classification model then evaluates the transformed code to ascertain its vulnerability status, categorising it as either vulnerable or non-vulnerable. If the line of code is predicted to be vulnerable, it is then transformed using the vectoriser linked with the loaded multi-class

[2]https://github.com/softwaresec-labs/ACVED
[3]https://github.com/softwaresec-labs/FedREVAN
[4]https://github.com/softwaresec-labs/Defendroid

model. This transformed code is subsequently fed into the multi-class model to predict the CWE-ID.

After the vulnerability status and the CWE-ID are predicted, the line of code undergoes processing methods akin to those employed in LVDAndro. This encompasses steps such as replacing comments and substituting user-defined strings. The resulting processed source code is then run through the Python Lime package, which is renowned for its XAI support. Lime is used to gain an understanding of the reasoning behind the predictions made by both the binary and multi-class models. This data is presented as prediction probabilities. The Lime package provides clarity on how individual words in the processed source code line contribute to the predictions, shedding light on their importance in both the prediction of vulnerability and the identification of the vulnerability category. Following this, the prediction outcomes are sent from the API as a JSON response and are then forwarded to the plugin.

7.4.1 Plugin Integration to Android Studio

The plugin is available as a JAR file, which can be downloaded from GitHub Repositories in three versions: 1) ACVED - an ensemble-based model, 2) FedREVAN - a federated neural Network-based model, and 3) Defendroid - a community-driven federated model. To smoothly incorporate any version of the plugin's JAR file into the most recent version of Android Studio, developers can simply adhere to the standard process for installing a third-party plugin in the Android Studio IDE. Moreover, to ensure compatibility with older versions of Android Studio, modifications can be made by adjusting the version specification as needed[5] in the plugin.xml file. Upon successful installation of the plugin, it provides suggestions related to vulnerabilities as balloon notifications. The plugin presents two options for detecting vulnerabilities that can be utilised while writing code. These options are visually depicted in Figure 7.7.

- **Quick Check**: This option entails conducting a comprehensive scan of the full source code file to detect any vulnerable source code.

  To activate the quick check option, the developer can go to *Tools* $\longrightarrow$ *Check Source Vulnerability*, or just use the shortcut *CTRL+ALT+E* within Android Studio. This option offers a fast way to pinpoint vulnerable lines of code and their corresponding CWE IDs in a single scan, providing a speedy evaluation of vulnerabilities.

[5] https://plugins.jetbrains.com/docs/intellij/android-studio-releases-list.html

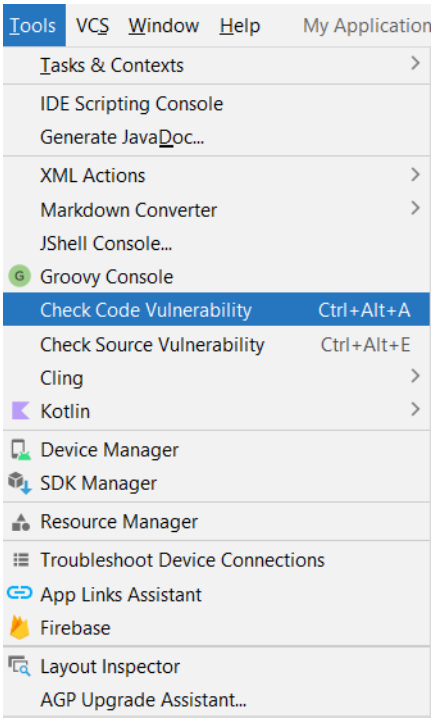


Figure 7.7: Plugin Integration with Android Studio in Tools Menu

- **Detailed Check**: In this option, the plugin assesses if a particular line of code is associated with any vulnerabilities.

  For the detailed check option, the developer can initiate it by navigating to *Tools* $\longrightarrow$ *Check Code Vulnerability*, or by using the shortcut *CTRL+ALT+A* when the cursor is on a particular line of code.

7.4.2 Plugin Usage

The balloon notification produced by the plugin communicates the results obtained from the API. Following a quick scan, developers receive a balloon notification that shows the status of any vulnerable code within the source file. If the scan does not detect any vulnerable code, a notification similar to Figure 7.8 is displayed.

Conversely, if segments of vulnerable code are detected, the notification, as depicted in Figure 7.9a, underscores the existence of such lines of code along with their respective CWE IDs. The notification is enlarged in Figure 7.9b for a more detailed view.

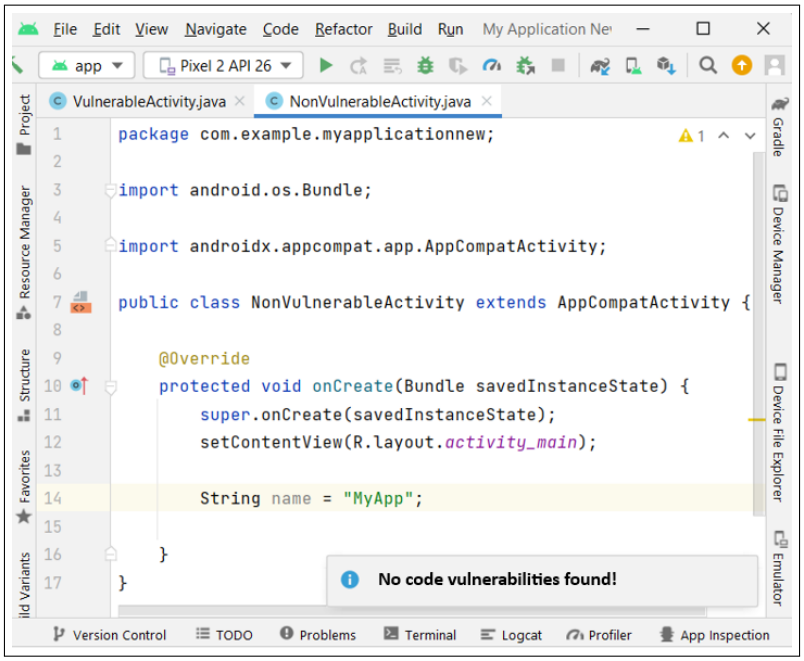


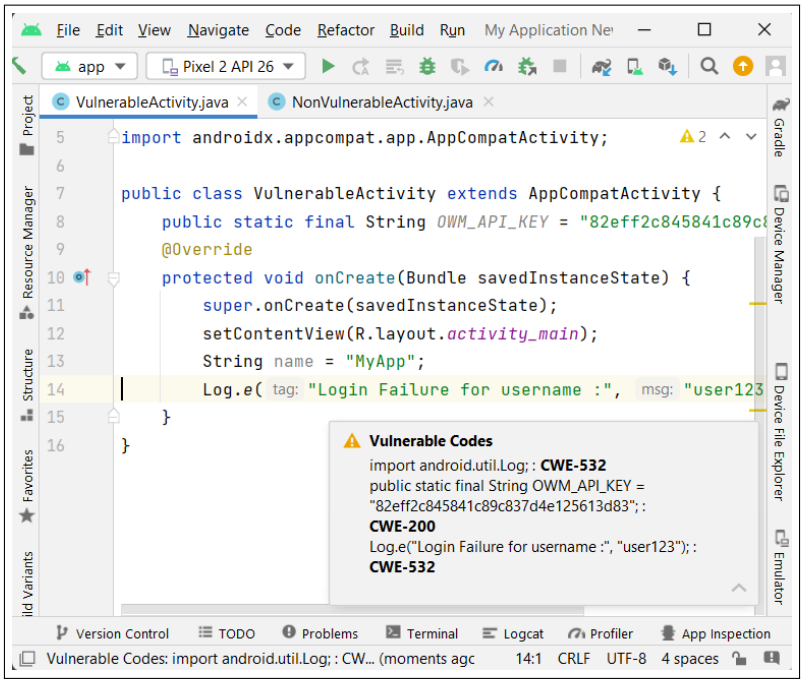
Figure 7.8: Quick Check Notification of Non-vulnerable Source File in Android Studio

During a detailed check, the developer will be alerted with a notification indicating the source code's vulnerability status. If the code is deemed secure and non-vulnerable, the developer will be notified with a message similar to the one in Figure 7.10. The focus here is on the code line *String name = "MyApp"*.

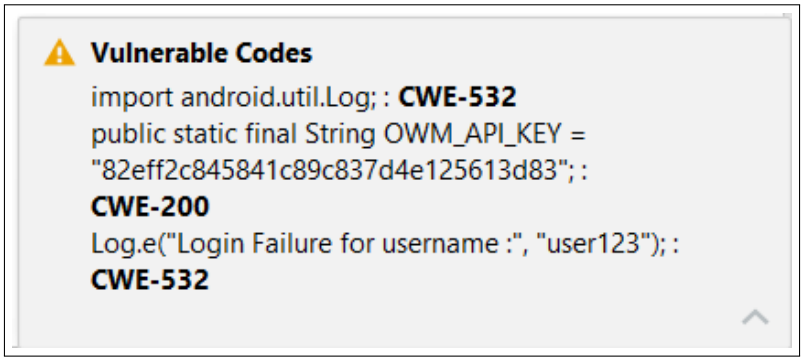
If the code at the current cursor position is detected as vulnerable, a balloon notification will appear as shown in Figure 7.11a. For a closer look at the specific balloon notification, refer to Figure 7.11b.

In this instance, the cursor is positioned on the line of code *Log.e("Login Failure for username:", "user123");*. This specific code is associated with CWE-532, a category that the model predicted correctly with a high confidence level of 0.99. Interestingly, within this prediction, the model recognised "Log" as the most impactful element, assigning it a significant probability of 0.53.

In terms of multi-class classification, the model showed a prediction probability of 0.99 for CWE-532, and it allocated a substantial contribution of 0.96 to the term "Log". This underscores the need for developers to be cautious when integrating log statements into production-level applications. These statements can potentially create vulnerabilities that attackers might exploit by examining log files. As a safeguard, developers can



(a) Notification in Android Studio



(b) Balloon Notification

Figure 7.9: Quick Check Notifications - Vulnerable Source File

employ encryption methods to produce log files in an encrypted format, rather than plain text, thereby enhancing security and reducing potential risks.

This notification provides an in-depth explanation of the security flaw, along with instructions on how to address it. It also contains data about the likelihood of predicting the status of the vulnerability (binary classification), the related CWE ID, and the prediction likelihood within the CWE category (multi-class classification prediction). Furthermore,

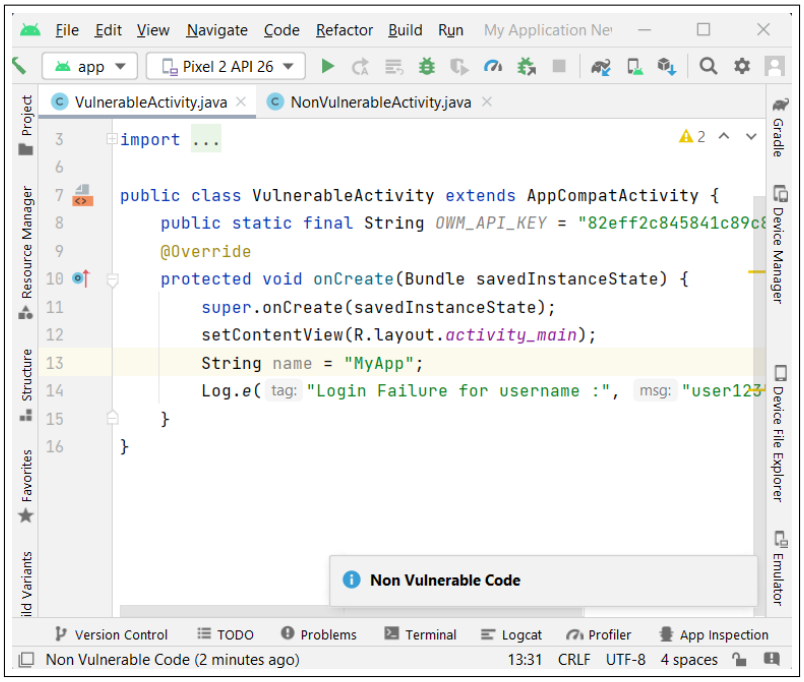
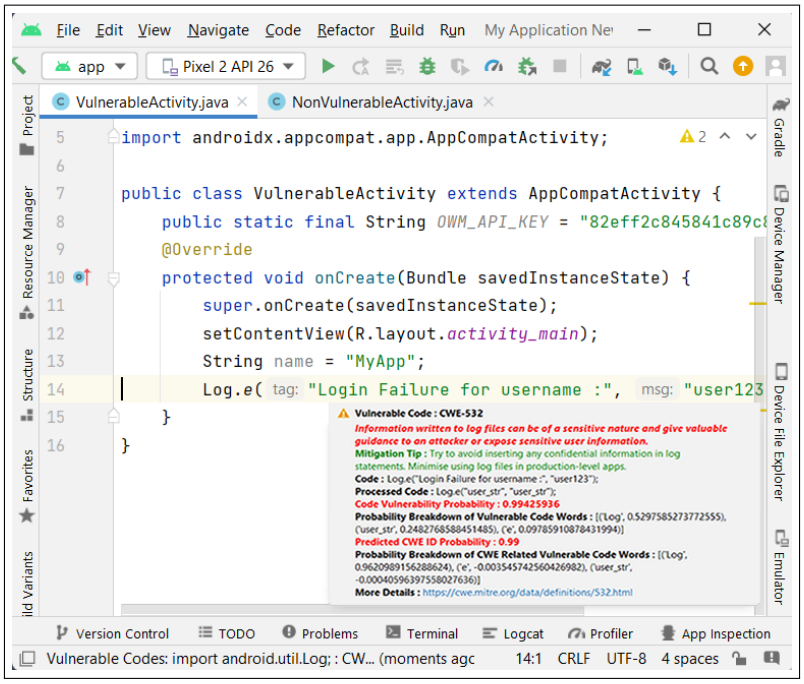


Figure 7.10: Detail Check Notification for a Non-Vulnerable Code Line in Android Studio

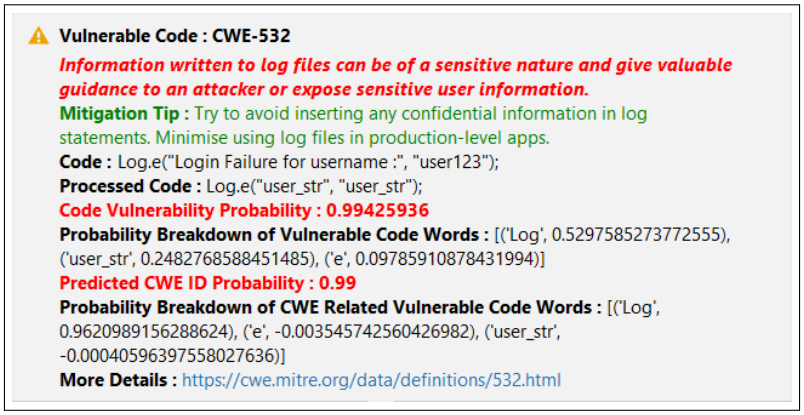
it explains how individual words contribute to the probability in binary and multi-class classification methods. The nature of the alert, be it informative or cautionary, depends on the seriousness of the security flaw. In addition, to provide a more complete understanding of the vulnerability, the plugin recommends ways to rectify it, typically by referencing the CWE repository [42]. The suggested mitigation recommendations can be further improved by leveraging large language models in future studies.

The Android Studio Event Log is in sync with these operations, as shown in Figure 7.12. This functionality assists developers in tracking the evolution of the source code when it comes to mitigating vulnerabilities.

Empowered with these suggestions and the associated prediction probabilities, developers are equipped to improve the security of their Android applications by tackling source code vulnerabilities. The capability to review the vulnerability evaluation gives developers the benefit of observing changes in prediction probabilities when certain lines of code are modified. This feature is advantageous in cases where full mitigation might not be possible, like instances where it is crucial to keep log file records for debugging, even in applications at the production level.



(a) Notification in Android Studio



(b) Balloon Notification

Figure 7.11: Detail Check Notifications - Vulnerable Source File

7.4.3 Assessing the Plugin Capabilities

The plugin has the ability to detect 10 categories of Common Weakness Enumeration (or 11 categories when including the *other* category). Each of these categories is associated with either a high or medium risk of exploitation, as detailed in [21]. The CWE-IDs that are covered include 89, 200, 276, 312, 532, 676, 749, 921, 925, and 939.

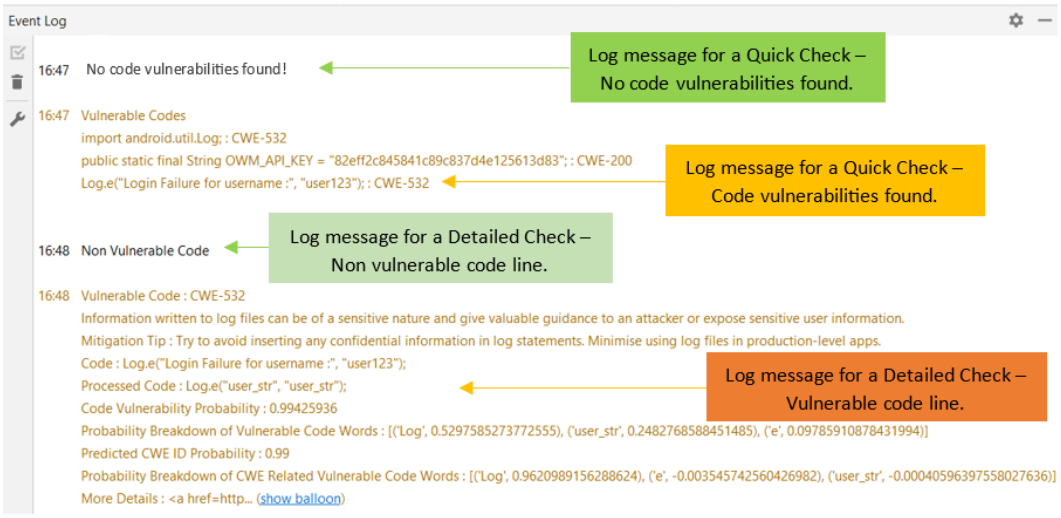


Figure 7.12: Plugin Notifications in Event Log

To evaluate the precision of the newly introduced Defendroid model and the plugin, a comparative study was carried out against the MobSF and Qark scanners, both of which played a role in creating the initial LVDAndro dataset. The assessment aimed to measure the precision of identifying vulnerable code within new data, involving a total of 2,216 lines of source code. This collection included a random selection of 604 lines of vulnerable code examples from the CWE repository, as well as 1,612 lines of non-vulnerable code taken from real applications. These lines of code were smoothly integrated into an Android app project and subsequently scanned using both MobSF and Qark Scanners. The same lines of code were then introduced to the Defendroid model via its Android Studio plugin. The comparative evaluation included metrics such as accuracy, precision, recall, and F1-Score, the results of which are summarised in Table 7.13.

Table 7.13: Accuracy Comparison of MobSF and Qark with Defendroid

Metrics	**MobSF**	**Qark**	**Defendroid**
Accuracy	0.91	0.89	**0.96**
Precision	0.93	0.92	**0.94**
Recall	0.95	0.93	**0.99**
F1-Score	0.94	0.92	**0.96**

Following the comparative study, it was clear that Defendroid surpassed MobSF and Qark in predicting vulnerabilities. Defendroid attained an impressive accuracy rate of

Table 7.14: Comparison of Average Time Taken to Analyse apps

App Categorise	MobSF	Qark	Defendroid
$Size < 1MB$	163s	123s	**100s**
$1MB \leq Size < 2MB$	181s	129s	**115s**
$2MB \leq Size < 4MB$	200s	165s	**122s**
$4MB \leq Size < 6MB$	277s	235s	**132s**
$6MB \leq Size < 8MB$	342s	372s	**162s**
$8MB \leq Size < 10MB$	397s	497s	**228s**
$10MB \leq Size < 12MB$	438s	543s	**259s**
$12MB \leq Size < 15MB$	451s	654s	**301s**
$15MB \leq Size < 20MB$	478s	729s	**313s**
$20MB \leq Size$	521s	752s	**329s**
Average	**344.8s**	**419.9s**	**206.1s**

96%, along with a precision score of 0.95, a recall rate of 0.99, and an F1-Score of 0.96. Importantly, Defendroid demonstrated excellence in lowering the false negative rate, thus reducing potential security threats linked to its predictions.

To assess the efficiency of vulnerability detection methods, researchers analysed fifty open-source Android projects from GitHub. They employed three tools: Defendroid (integrated with Android Studio), MobSF, and Qark. The apps were categorised by size, with five apps per category. The average analysis times for each method were measured, and the experiments were conducted on a Windows OS environment with a Core i5 processor and 16GB RAM. The results indicate that Defendroid outperforms MobSF and Qark in terms of speed. Defendroid detected vulnerabilities in just 206.1 seconds, compared to MobSF's 344.8 seconds and Qark's 419.9 seconds. Notably, this performance comparison focused on completed applications due to limitations in existing vulnerability scanners. One of Defendroid's key advantages is that it does not require building the entire application.

When using the Defendroid plugin within the Android Studio IDE, developers can seamlessly continue their standard coding process without disruption. The plugin provides real-time predictions for individual code lines in less than 300 milliseconds. This evaluation was conducted in an Android Studio Chipmunk version running on a Windows OS environment with a Core i5 processor and 16GB RAM. However, it is worth noting that initiating the Defendroid API prior to execute the plugin, takes between 3 to 10 seconds in the same environment. Despite this initialisation time, app developers do not need to allocate any additional effort or time to obtain these valuable real-time prediction results.

Defendroid's functionalities were also contrasted with various well-known tools and techniques employed for vulnerability detection, as shown in Table 7.15 and Table 7.16.

Moreover, the model's performance is expected to continually improve and achieve peak levels as this blockchain-based, privacy-preserved federated learning environment becomes available to a wide range of clients and miners, from solo app developers to app development corporations.

Table 7.15: Key Features Comparison of Defendroid with Other Popular Vulnerability Detection Tools

Tool/Method	**Detection Capabilities and Computational Requirements**	**Integration Capabilities and User Friendliness**	**Other Features**
Snyk [198]	* Claiminig to be near zero false positive * Need to be executed parallel	* Can be integrated with IDEs * Unable to provide reasons for detected vulnerabilities	* Limited features are available in free version
ImmuniWeb MobileSuite [199]	* Claiminig to be zero false-positive * Need to be executed parallel	* Can be integrated with SDLC * Provides actionable remediation guidelines	* Supports for mobile app and backend testing
Drozer [200]	* Interact with the Dalvik VM and other endpoints of the app to find vulnerabilities * Unable to detect vulnerabilities in real-time	* Unable to integrate with Android development environments * Less user friendly due to the command line based approach	* Supports penetration testing * Search for security vulnerabilities in apps * Free and open source
Astra Pentest [201]	* Performs over 8000 test cases to aid in the detection of vulnerabilities * Can identify misconfiguration errors in code or build settings	* Lack of ability to detect code level vulnerabilities at early stages * Need to run it as a separate program	* Supports for automated and manual penetration testing * Not available for free
ACVED [21]	* Ensemble AI model with 95% accuracy * 0.95 F1-Score in binary model and 0.93 F1-Score in multi-class model * Model re-training time is high * API initiation time is high	* Can be integrated with Android Studio * Explain vulnerabilities	* Training the model in a central location * Relies solely on the LVDAndro dataset * Free and open source
Defendroid	* Neural Network based model with 96% accuracy * 0.96 F1-Score in both binary and multi-class models * API initiation time and model re-training time is low	* Can be integrated with Android Studio * Community driven and easily extendable * XAI based vulnerability mitigation * Work in progress to improve the usability of the plugin	* Privacy-preserved block chained federated learning model * Limited capability to detect complex vulnerability patterns * Free and open source

Table 7.16: Features Summary of Defendroid with Other Popular Vulnerability Detection Tools

Feature	MobSF [234]	Qark [161]	Snyk [198]	Immuni Web [199]	Drozer [200]	Astra [201]	ACVED [21]	**Defendroid**
Detect Android code vulnerabilities	✓	✓	✓	✓	✓	✓	✓	✓
Detect vulnerabilities in real-time by integrating with IDEs	-	-	✓	-	-	-	✓	✓
Detect vulnerabilities line by line	-	-	✓	-	-	-	✓	✓
Detect vulnerabilities in whole source code	✓	✓	✓	✓	✓	✓	✓	✓
Provide suggestions to mitigate vulnerabilities	-	-	✓	-	-	-	✓	✓
Explain the reasons for vulnerabilities	-	-	-	-	-	-	-	✓
Preserve the privacy of source code	-	-	-	-	-	-	-	✓
Free and open source	✓	✓	-	-	✓	-	✓	✓
Able to run alongside the development platform	-	-	✓	-	-	-	✓	✓
AI-based backend	-	-	✓	-	-	-	✓	✓
Community driven and easily scalable	-	-	-	-	-	-	-	✓

7.5 Chapter Summary

This chapter examined the model's enhancements using a community-driven federated learning approach and the implementation of differential privacy, resulting in a more privacy-centric model that can draw more training data from various clients. It also discussed real-time vulnerability mitigation using an Android Studio plugin, which uses the top-performing AI-based model as its back-end. This conversation underscored the practicality of how developers could employ the model. As targeted in the preceding chapter, it was feasible to enhance the model's performance, achieving an accuracy of 96% for both binary and multi-class classifications using this community-driven federated learning approach. Furthermore, the model has the potential to improve data privacy with the use of the proposed model. By making substantial contributions in these areas, it also addressed the question RQ4 and achieved the objective RO5 as specified in chapter 1.

Chapter 8

Case Study: Use of the Defendroid Plugin for Android Code Vulnerability Detection

This chapter explores the utilisation of the "Defendroid", newly developed AI-powered Android Studio plugin, which is engineered to identify potential vulnerabilities in Android code. Initially, a need analysis survey was conducted to identify the necessity of such a plugin, and then its efficiency was evaluated by involving Android application developers in its operation. The findings are subsequently collected, structured into a case study, and showcased.

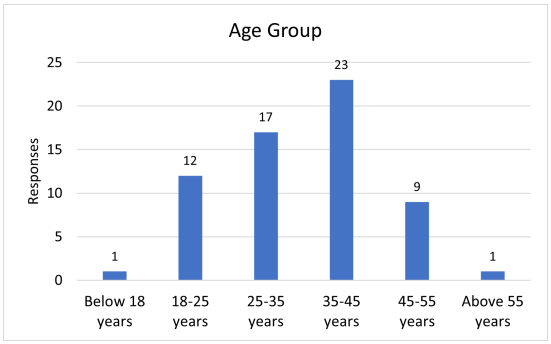
8.1 Need Analysis for Android Code Vulnerability Detection Method

The literature review uncovered a lack of tools capable of detecting security flaws in real-time during the development of Android applications. To validate this observation, a preliminary action was undertaken to conduct a needs analysis survey. This survey involved 63 Android application developers employed by app development firms, aiming to ascertain whether security considerations were incorporated into their app development workflows. These participants were chosen based on a variety of personal and professional connections and identified via professional social networks. Since individual responses were not disclosed and the analysis was conducted on an aggregate basis, there were no

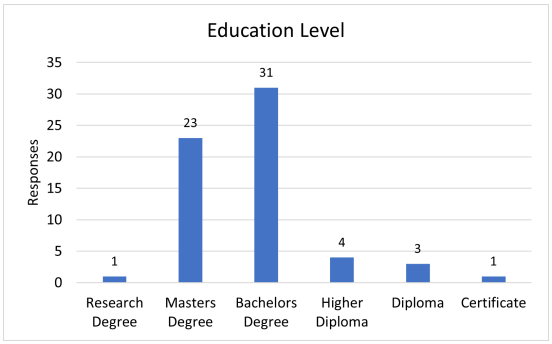
ethical concerns associated with the survey. Participants were informed of this approach, and the survey was conducted after obtaining their consent.

8.1.1 Characteristics of the Selected Android Developers

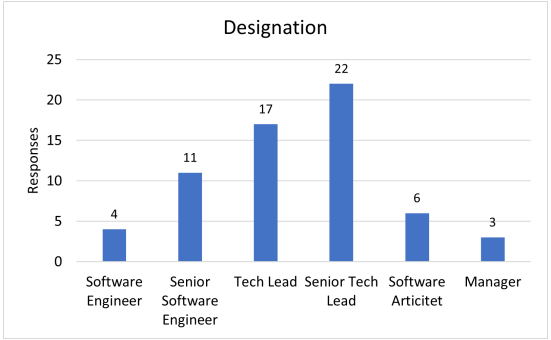
An initial survey was conducted before the use of the plugin to comprehend the traits of the Android developers who took part in the evaluation of the plugin. The questions in the survey are included in Appendix A. Examining the demographic characteristics of the developers is advantageous in determining their proficiency in software development, Android app creation, and secure development. Additionally, factors such as age group, educational level, and the nature of the companies they work for are crucial in ensuring that the selected group of Android developers accurately represents the majority of the intended user base for the plugin. This is because the plugin will primarily be used by developers with similar characteristics, including the type of company and their job roles. The corresponding responses to those questions are depicted in Figure 8.1 and Figure 8.2.



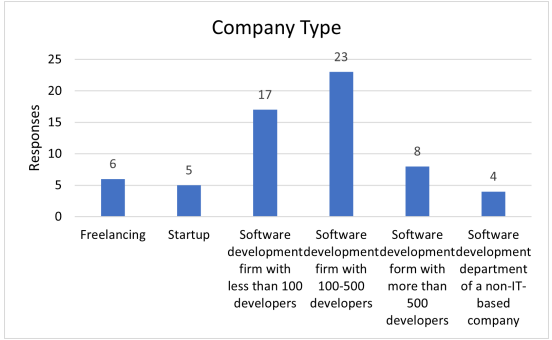
(a) Ch-Q1 - Age Group



(b) Ch-Q2 - Education Level



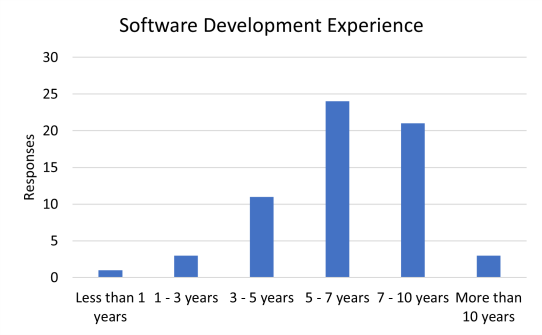
(c) Ch-Q3 - Designation



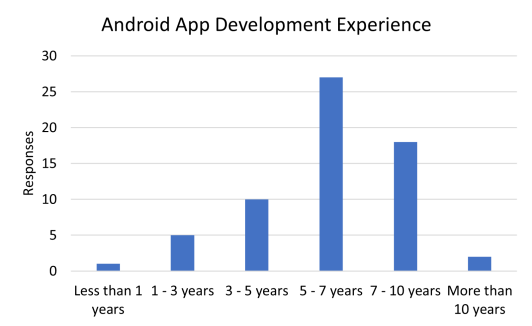
(d) Ch-Q4 - Company Type

Figure 8.1: Characteristics Analysis Q1 to Q4

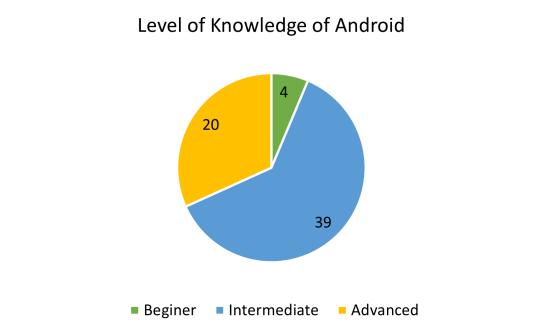
The responses indicated that the selected developers represent a varied group of Android developers, with most being in the active working age of 18-45 years and occupying high-ranking roles. On average, the group has 6 years of experience in software development and Android app creation, employed in various medium to large-scale software firms.



(a) Ch-Q5 - Software Development Experience

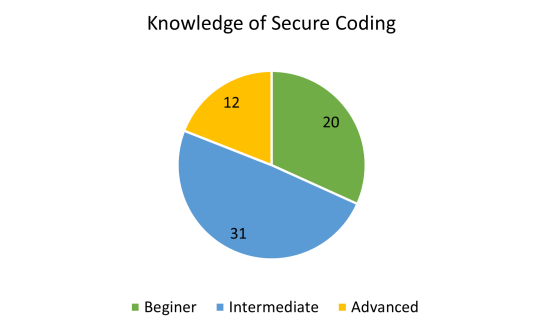


(b) Ch-Q5 - Android Development Experience

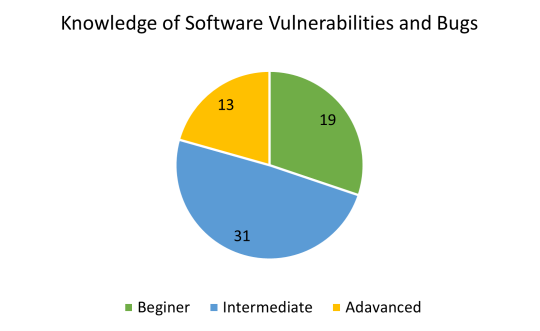


(c) Ch-Q7 - Android Knowledge

(d) Ch-Q8 - Software Security Training



(e) Ch-Q9 - Secure Coding Knowledge



(f) Ch-Q10 - Vulnerabilities and Bugs

Figure 8.2: Characteristics Analysis Q5 to Q10

According to the response, it was also identified that the developers have notable proficiency in software development, with the majority having an intermediate understanding of Android, secure coding practices, and vulnerability detection.

8.1.2 Secure Coding Practices of the Selected Android Developers

Developers were posed with a question (Sc-Q1) concerning whether they take into account secure coding practices during application development. The responses highlighted that a substantial percentage of developers, precisely 54%, do not integrate secure coding guidelines into their app creation processes, as illustrated in Figure 8.3.

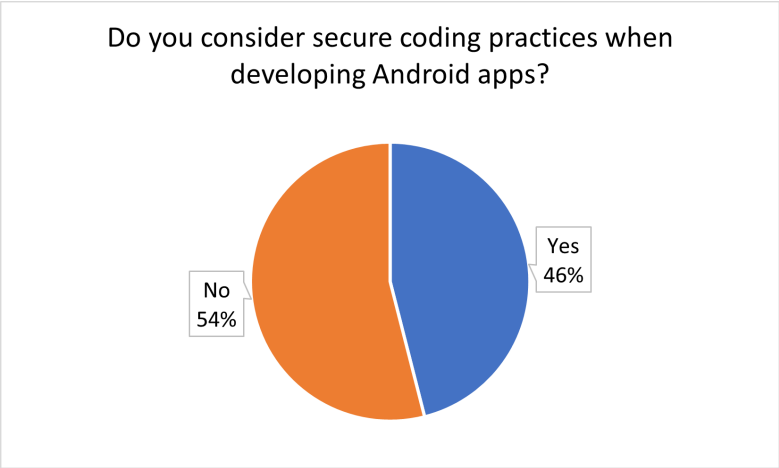


Figure 8.3: Consideration of Secure Coding

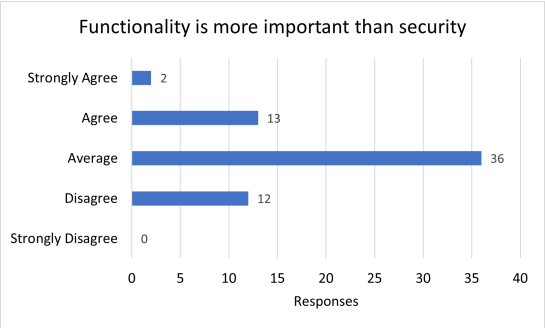
In order to comprehend why most developers do not incorporate secure coding principles, the participants were requested to rate the factors that influence their minimal attention to secure coding, on a 5-point Likert scale (Strongly Agree, Agree, Average, Disagree, Highly Disagree) for four statements (SC-Q2 to SC-Q5). The outcomes related to these factors are visually represented in Figure 8.4.

Sc-Q2: Functionality is more important than security.

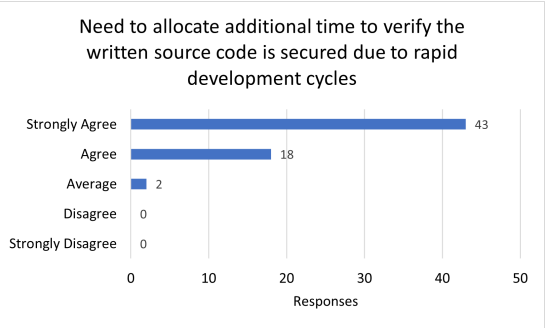
Sc-Q3: Need to allocate additional time to verify the written source code is secured due to rapid development cycles.

Sc-Q4: Manual verification requires additional resources, including domain experts.

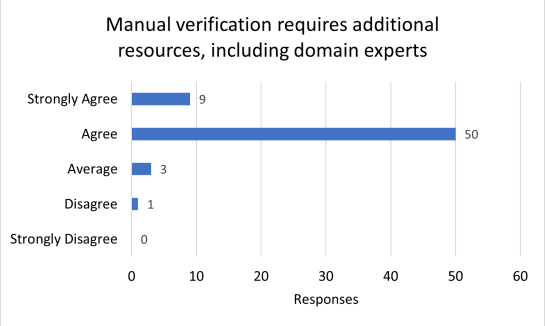
Sc-Q5: There is a lack of supportive tools to automate the security checking process.



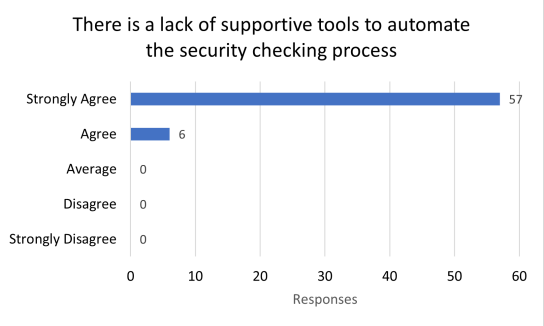
(a) Sc-Q2 - Functionality vs Security



(b) Sc-Q3 - Additional Time



(c) Sc-Q4 - Additional Resources



(d) Sc-Q5 - Lack of Tool Support

Figure 8.4: Reasons for Underestimating Secure Coding

After examining the responses, it was evident that a significant number of app developers assign equal weight to both functionality and security, with 36 responses falling under the *Average* category. In addition, a substantial majority of developers, making up 68%, strongly concur with allocating extra time for scrutinising code from a security perspective. There is a mutual agreement among developers about the importance of involving domain experts such as security testers and ethical hackers in the development process, particularly when manual security verification is required. This is due to the potential deficiency of expertise among developers in identifying source code vulnerabilities and implementing secure coding practices. Furthermore, 90% of the respondents strongly agree that the absence of adequate tool support is a primary reason for not paying sufficient attention to or underestimating security aspects during app development. Consequently, it was concluded that incorporating a highly accurate automated vulnerability detection model into the development pipeline is essential. Therefore, it was determined that the employment of the high-precision Android code vulnerability detection model integrated as a plugin with Android Studio is crucial.

8.2 Developer Feedback

The group of 63 Android app developers received the plugin containing the latest model. They were given instructions on installing and running the plugin, which they then used while coding in a real-time Android Studio environment. After this trial period of two months, a survey was conducted to gather feedback on the plugin's performance and capabilities. The developers expressed their satisfaction levels using a 5-point Likert scale (ranging from Excellent to Very Poor). Additionally, domain experts in security testing and ethical hacking, collaborated with the developers to manually analyse and verify the prediction results generated by the plugin. The feedback was evaluated based on six criteria as presented graphically in Figure 8.5.

DF-Q1: The easiness of integration and configuration of the plugin with Android Studio.

DF-Q2: The accuracy of vulnerable code and its CWE category predictions.

DF-Q3: The efficiency of providing vulnerability prediction results.

DF-Q4: The easiness of using the plugin with the usual app development process.

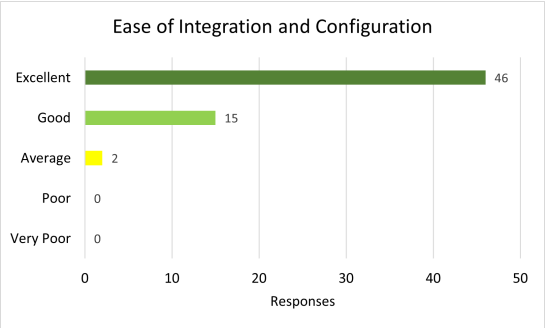
DF-Q5: The usefulness of mitigation suggestions provided by the plugin.

DF-Q6: The look and feel of the notification of plugin with prediction results.

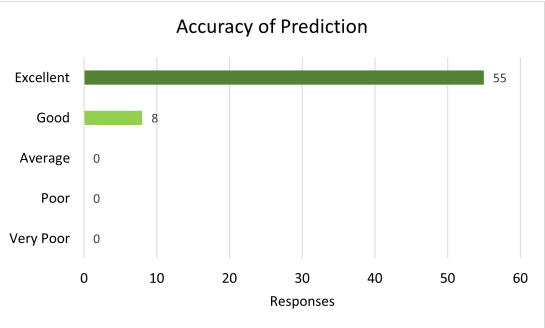
The results of the survey revealed that a substantial majority—87% of app developers expressed high satisfaction with the accuracy of the plugin's predictions. Additionally, 94% of the developers rated the prediction efficiency as excellent. Furthermore, an impressive 92% of developers reported high satisfaction with the plugin's overall usefulness and the quality of its mitigation recommendations.

However, the survey highlighted opportunities for improving the plugin's usability and integration features. Approximately 70% of developers expressed high satisfaction with the ease of use, leaving room for enhancement. Additionally, there is a potential to enhance the plugin's visual aesthetics. Most developers rated its look and feel as average. This feedback is valuable for enhancing the plugin's appeal. One possible approach involves integrating mitigation suggestions similar to IDEs' syntax error indicators. This would provide more intuitive recommendations by highlighting issues directly rather than relying on balloon notifications.

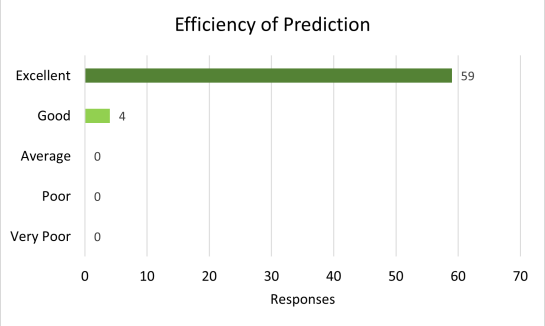
In addition to the criteria mentioned above, the overall satisfaction with the plugin (DF-Q7) was also assessed by developers through the same survey. They were asked to



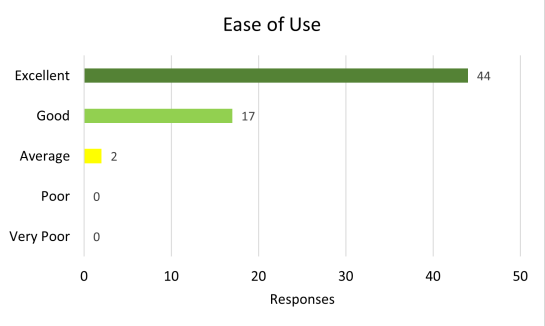
(a) DF-Q1 - Ease of Integration, Configuration



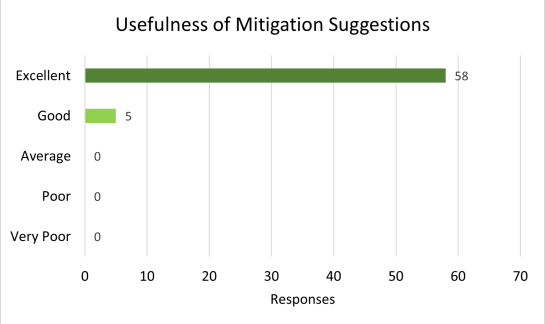
(b) DF-Q2 - Accuracy of Prediction



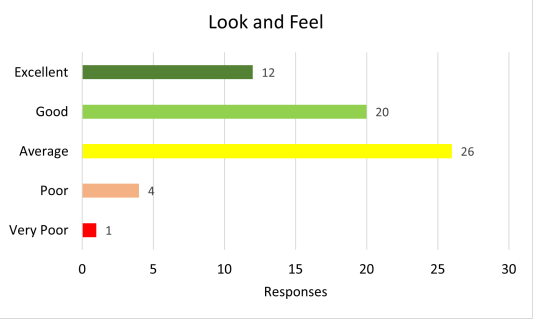
(c) DF-Q3 - Efficiency of Prediction



(d) DF-Q4 - Ease of Use



(e) DF-Q5 - Mitigation Suggestions



(f) DF-Q6 - Look and Feel

Figure 8.5: Developer Satisfaction

evaluate using the same 5-point Likert scale, and the responses are depicted in Figure 8.6.

Despite the identified areas for improvement, the overall satisfaction rate remains remarkably high. 79% of developers expressed a high degree of satisfaction, and an additional 21% reported a good level of satisfaction. With further development and refinement,

Figure 8.6: Overall Satisfaction of the Plugin

the plugin has the potential to gain wider adoption within the developer community for mitigating Android source code vulnerabilities.

Beyond the previously mentioned queries, developers were invited to share their views on the plugin. Since responding to this was optional, not all developers provided feedback. The individual comments and feedback are mentioned in Appendix B.

Developers have praised the plugin as a valuable tool that addresses a real challenge in app development by reducing code vulnerabilities. They appreciate its seamless integration into the development process, its remarkable prediction time, and its ability to provide robust support for addressing vulnerabilities. The plugin's real-time assistance, instant identification of vulnerabilities, and its free availability are also highly valued. Developers suggest making this plugin official with Android Studio's endorsement and expanding its functionality to other programming languages. They also proposed improvements such as identifying vulnerabilities as soon as a line of code is typed, starting the backend model as soon as Android Studio opens, providing a feature to scan the whole project at once, and providing additional guidance like suggesting vulnerability-free code. These feedback and enhancement proposals align with previous findings and can be considered for future improvements.

8.3 Chapter Summary

This chapter explores the perspectives of Android app developers on using the newly developed AI-based plugin to address vulnerabilities during app development. Initially, a needs analysis survey was conducted to gather user opinions on the use of a tool for secure software development. This analysis also took into account the characteristics of the selected audience. After the analysis, developers were instructed to download the plugin, install it in Android Studio, and use it during app development. Following a trial period, developers were asked to evaluate the plugin's capabilities. Upon analysing the developers' feedback, it was found that they were extremely satisfied with the plugin and suggested a few improvements to enhance its functionality.

Chapter 9

Conclusion

This chapter serves as a concise summary of the extensive work carried out throughout this research. It revisits the findings that were discussed in depth in the previous chapters, providing a summary of these results. This includes an evaluation of the strengths of the research, highlighting its successful aspects, and limitations of the proposed method. Additionally, potential avenues for future exploration are also considered in this chapter.

9.1 Discussion and Summary

The importance of incorporating secure coding practices cannot be overstated when it comes to the development of Android applications. This research presents a novel approach that is centered around privacy, driven by the community, and powered by artificial intelligence. This approach is designed to identify potential vulnerabilities with a high degree of accuracy during the process of writing the source code. The model developed as part of this research is seamlessly integrated into the Android Studio IDE in the form of a plugin. This integration allows Android developers to leverage the model's capabilities to detect potential vulnerabilities in their code in real-time as they write it. This real-time detection can significantly enhance the security of the applications being developed. The methodology employed in this doctoral research, which led to the development of this innovative approach and model, has been discussed in detail in chapter 4. This methodology provides a comprehensive understanding of the steps taken and the considerations made during the course of this research. It serves as a roadmap for the research process, shedding light on how the objectives were achieved.

The initial step in this doctoral research was to pinpoint a specific area of study that had not been thoroughly explored, which was accomplished by conducting an extensive review of the existing literature. This process was not a one-time event, but rather a continuous effort that spanned the entire duration of the research. During this process, a comprehensive exploration of existing techniques for application and code analysis was undertaken. This exploration was not limited to understanding these techniques but also involved investigating potential applications of machine learning and deep learning methodologies. The goal was to identify innovative ways to enhance the effectiveness and efficiency of code analysis. In addition to exploring these techniques, the research also delved into several other areas. The availability of datasets, which are crucial for training and testing machine learning and deep learning models, was one such area. The potential for applying explainable AI, which provides insights into how these models make their decisions, was another area of interest.

The research also looked into the possibility of using federated learning, a technique that allows for decentralised learning across multiple devices or servers. This technique allows for model training without the need to share raw data. The use of differential privacy, a method to enhance the privacy of training data, was also explored. Finally, the potential application of blockchain technology, a community-driven approach, was investigated. These various aspects of the research were discussed in detail in chapter 2 and chapter 3. These chapters provide a comprehensive overview of the background information and literature that informed the research process and methodology.

The existing datasets posed several limitations that made them unsuitable for the development of a high-accuracy, AI-based model capable of detecting vulnerabilities in Android code in real-time. This necessitated the creation of a new, more suitable dataset. As a significant contribution to this doctoral research, a novel dataset, referred to as LVDAndro, was developed. This dataset was created using a hybrid scanning approach that combined the capabilities of several high-accuracy Android app vulnerability scanning tools. Over 15,000 Android apps were scanned and the vulnerable source code lines were then labelled according to the CWE categories. This process resulted in the LVDAndro dataset, which contains over 2,000,000 unique code samples. The creation of the LVDAndro dataset represents a significant step forward in the field of Android app security. It provides a valuable resource for researchers and developers alike, enabling them to better understand and address the vulnerabilities that can arise in Android code. The detailed statistics of the LVDAndro dataset, including the distribution of code samples

across different CWE categories and other relevant information, were discussed in chapter 5. This chapter provides a comprehensive overview of the dataset, shedding light on its structure, contents, and potential applications in the field of Android app security.

The practicality of the LVDAndro dataset for identifying vulnerabilities in Android code was initially showcased through a proof-of-concept using AutoML. This demonstration confirmed that the LVDAndro dataset could be effectively used to train AI-based models for binary and multi-class classification tasks. To further enhance the capabilities of the model, an ensemble model was constructed. This model incorporated several machine learning algorithms that are commonly used for this type of task. The ensemble approach combines the strengths of these different algorithms to improve the overall performance and accuracy of the model. In addition to the ensemble model, a shallow neural network model was also developed to further optimise the model's performance. Model pruning techniques were employed as part of this process. These techniques help to simplify the model and reduce over-fitting by removing unnecessary parts of the neural network. However, merely predicting vulnerabilities is not sufficient for meaningful mitigation. Therefore, explainable AI was integrated into the model's API. This allows users to understand the reasoning behind the model's predictions, providing valuable insights that can aid in the mitigation of identified vulnerabilities. These experiments and their outcomes were discussed in detail in chapter 6. This chapter provides a comprehensive overview of the AI model development process, the techniques employed, and the results achieved.

One of the significant challenges in developing AI-based models, including the specific research problem addressed in this doctoral research, is the scarcity of training data. To overcome this hurdle and to engage more clients who are willing to participate in the model training process, the concept of federated learning was employed. To further enhance the model's validation process and encourage community participation, a blockchain-based environment was integrated. Blockchain technology, known for its transparency and immutability, can provide a trustworthy platform for model validation. In addition, to bolster the privacy of the model, differential privacy was applied within this blockchain-based federated learning environment. This ensures that the privacy of individual data contributors is protected, even when a significant amount of aggregate data is shared. These innovative approaches significantly contributed to the research process. The resulting model achieved an impressive accuracy of 96% and an F1-Score of 0.96 for both binary classification and multi-class classification.

The final model, combined with XAI, was integrated into an API. This API was then incorporated as the backend of a newly developed plugin for the Android Studio IDE. This integration allows developers to detect vulnerabilities in real-time as they write code in the Android Studio IDE, thereby enhancing the security of the applications they develop. These steps and their implications were discussed in depth in chapter 7. To assess the practicality and effectiveness of the newly developed plugin in real-world scenarios, a user survey was conducted among Android application developers. This survey, detailed in chapter 8, served as a case study, providing valuable insights into the use of the model and plugin in actual Android app development. The feedback from the developers was overwhelmingly positive. They expressed high appreciation for the plugin's value, particularly praising its excellent performance in detecting vulnerabilities in real-time as they wrote code in the Android Studio IDE. This real-time detection capability was seen as a significant advantage, enhancing the security of the applications they developed. However, the developers also provided constructive feedback, pointing out some areas that could benefit from further refinement. This feedback is invaluable, as it provides direction for future improvements to the plugin, ensuring it continues to meet the evolving needs of Android developers. This iterative process of development and feedback is crucial in ensuring the continued relevance and effectiveness of the plugin in enhancing Android app security.

9.2 Revisiting Objectives

In this section, research objectives established in chapter 1 were revisited. A summary of how each of these objectives has been tackled in the course of this doctoral research is discussed.

RO1: To recognise and evaluate current methods for detecting vulnerabilities in Android source code.

This objective accomplished as detailed in chapter 2 and chapter 3. The literature review revealed the absence of a highly accurate, privacy-focused AI technique for identifying vulnerabilities in Android code within a real-time development setting. The scarcity of properly labelled datasets was a significant issue for such methods.

RO2: To generate a labelled dataset of vulnerabilities in Android source code through an extensive literature survey.

To tackle the issue of the absence of a properly labelled dataset for Android code

vulnerabilities, a new dataset called LVDAndro was created. The vulnerable codes were marked according to CWE IDs and included source code scanned from more than 15,000 actual Android applications. This objective was accomplished as detailed in chapter 5.

RO3: To develop an accurate AI-based technique for real-time detection of source code vulnerabilities in Android app development, utilising the dataset generated from RO2.

While achieving this objective, the practicality of the LVDAndro dataset was showcased in chapter 6 as part of a proof-of-concept, thereby accomplishing this objective. Initially, an AutoML-based approach was used, followed by the application of various machine learning and deep learning techniques to detect code vulnerabilities as the code is being written.

RO4: To devise a strategy for offering suggestions to mitigate code vulnerabilities, utilising the method developed in RO3.

Merely offering a prediction is not enough for developers to address vulnerabilities. Developers require guidance on how to evade or lessen the vulnerability's impact. As such, the newly developed AI-based method for detecting Android code vulnerabilities was enhanced with XAI. This provides developers with prediction probabilities, helping them comprehend the rationale behind a specific prediction and identify potential mitigation strategies. This objective was fulfilled in chapter 6.

RO5: To improve the training data while preserving privacy and enhancing the detection capabilities of the model developed in RO3.

The introduction of a variable differential privacy technique, combined with blockchain-based federated learning, enhanced the model's performance while preserving the privacy of the training data from a wide range of clients. This objective was accomplished as detailed in chapter 7. The final, best-performing model which has an accuracy of 96% and an F1-Score of 0.96 for both binary and multi-class classification was subsequently incorporated as the backend of an Android Studio plugin and made available to Android developers for use in their regular app development workflow. The plugin's capabilities were assessed with the assistance of a group of Android app developers, as discussed in chapter 8.

9.3 Limitations

Developing secure Android applications is crucial for attracting a large user base. Therefore, protecting its code using a highly accurate AI-based method is a prevailing industry trend, given the surge in research activities involving new technologies. While this research presents significant advancements, there are opportunities for further enhancement that could address some potential limitations and weaknesses.

Although the LVDAndro dataset is extensive, there is room to enhance its representation by diversifying the selection of apps and vulnerabilities. Expanding the dataset to cover a broader range of real-world scenarios could help mitigate any inherent biases in the data collection process. As the dataset continues to grow in complexity, there is potential to develop more efficient methods for maintaining and updating it. These methods could better reflect new vulnerabilities and evolving coding practices, ensuring the dataset remains relevant and robust against emerging threats. Further validation and optimisation of the model could be beneficial to ensure consistent performance across diverse real-world applications. Expanding testing across various types of Android apps could help refine the model's accuracy in different environments.

Integrating the plugin into Android Studio is an exciting development, and optimising it to minimise performance overhead would be a valuable next step. Addressing compatibility issues with different versions of Android Studio and other plugins could further streamline its adoption in large-scale projects. The positive feedback from the user survey is encouraging, and there is an opportunity to refine the plugin further to enhance usability. Providing additional resources or training could ease the learning curve and increase adoption among developers who might be resistant to workflow changes. Conducting longitudinal studies or extensive real-world testing would provide deeper insights into the long-term effectiveness of the plugin. This would validate its sustained impact on improving code security and reducing vulnerabilities over time.

While the integration of XAI is a strong feature, there is potential to improve the clarity and accessibility of the explanations provided. Simplifying these explanations could make them more actionable for developers of all expertise levels, thereby increasing the practical utility of XAI. Enhancing the robustness of privacy measures within the federated learning framework could alleviate concerns about data leakage or misuse. Strengthening differential privacy mechanisms could further safeguard sensitive information in this collaborative environment. To keep pace with the rapid evolution of security threats and Android development practices, establishing a framework for regular updates and

continuous learning would be crucial. This approach would ensure that the model and dataset remain effective and relevant in an ever-changing landscape.

9.4 Possible Future Directions

Throughout this research process, several potential future directions have been identified that could help address some of the limitations of the proposed method.

9.4.1 Increase the Detection Capabilities of the Model

The existing model has attained an accuracy rate of 96% for both binary and multi-class classification, with an F1-Score of 0.96. However, due to the limited number of samples for some CWE categories in the LVDAndro dataset, the current model can accurately identify 10 CWE categories. At present, apart from these 10 CWE IDs, the remaining vulnerable codes will be classified under the 'other' category. Although the current 10 CWE categories represent the most commonly observed vulnerabilities, it would be beneficial to enhance the model's ability to detect a broader range of vulnerabilities. This could be achieved by expanding the model's training capabilities through the integration of a diverse set of clients and adding multiple vulnerability scanners.

The domain of large language models and generative AI is rapidly progressing. The existing model for detecting vulnerabilities in Android code can benefit from these advancements. By harnessing the capabilities of these evolving technologies, the model's ability to identify vulnerabilities can be significantly enhanced. This could potentially lead to more accurate and comprehensive detection of security flaws in Android applications, thereby improving the overall security of these apps. Large language models can also be employed to augment the training dataset, thereby enhancing it. This could potentially overcome the current model's limitation of detecting only 10 categories of vulnerabilities.

9.4.2 Customise the Model to be Trained Using a Public Blockchain While Providing Attractive Rewards

Owing to infrastructure costs and development time constraints, the model currently utilises a privately customised blockchain-based federated learning environment for training. Although the current model predicts Android code vulnerabilities with high accuracy, the expansion of the model using the existing architecture is identified as a limitation during the research process. This is because only invited clients can collaborate

with the model training due to the network being private. The blockchain-based environment can be customised to operate on a public blockchain network such as Ethereum or Hyperledger Fabric. This could potentially engage more clients and also introduce a proper incentive rewarding mechanism, thereby enhancing the detection capabilities of the model.

9.4.3 Complex Vulnerability Patterns Detection

The proposed model can accurately pinpoint a multitude of code vulnerabilities during the development of Android apps. However, the current model lacks the ability to detect intricate patterns of vulnerabilities such as logical vulnerabilities and other dynamically occurring vulnerabilities. By integrating a hybrid analysis feature, the model may be further improved to detect them. Consequently, app developers will have the advantage of detecting Android app vulnerabilities with high precision at the design stage, thereby further securing the apps from vulnerabilities that can arise. Additionally, when the model is being trained in a federated learning environment, runtime analysis features related to vulnerabilities can also be identified and enhanced.

9.4.4 Increase the Model Privacy and Security more with Homomorphic Encryption and Secure Multi-Party Computation

Homomorphic Encryption [235] and Secure Multi-Party Computation [236] are two powerful techniques that can greatly enhance the privacy and security of an Android code vulnerability detection model. Homomorphic Encryption allows for computations to be carried out on encrypted data without the need for decryption. This means that the model can process and analyse encrypted data while preserving the privacy of the original data. This is especially useful in a federated learning environment where data privacy is of paramount importance. Secure Multi-Party Computation is a method that allows multiple parties to compute a function over their inputs while keeping those inputs private. In the context of Android code vulnerability detection, this can be used to aggregate model updates from multiple clients in a privacy-preserving manner. Each client can contribute to the model training without revealing their private data. This not only protects the privacy of the data but also ensures the integrity and confidentiality of the model itself. However, it is important to note that implementing these techniques may increase the computational complexity and could impact the performance of the model. Therefore, careful consideration and optimisation are required when integrating these techniques into the model.

Bibliography

[1] Yang K, Miller P, Martinez-Del-Rincon J. Convolutional Neural Network for Software Vulnerability Detection. In: 2022 Cyber Research Conference - Ireland (Cyber-RCI); 2022. p. 1-4. Available from: https://doi.org/10.1109/Cyber-RCI55324.2022.10032684.

[2] Statista. Average number of new Android app releases via Google Play per month from March 2019 to Feb 2024; 2024. Available from: https://www.statista.com/statistics/1020956/android-app-releases-worldwide/.

[3] Garg S, Baliyan N. Comparative analysis of Android and iOS from security viewpoint. Computer Science Review. 2021;40:100372. Available from: https://doi.org/10.1016/j.cosrev.2021.100372.

[4] Krasner H. The cost of poor software quality in the US: A 2020 report; 2021. Available from: https://www.it-cisq.org/cisq-files/pdf/CPSQ-2020-report.pdf.

[5] Ghaffarian SM, Shahriari HR. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Computing Surveys (CSUR). 2017;50(4):1-36. Available from: https://doi.org/10.1145/3092566.

[6] Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, et al. Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE international conference on machine learning and applications (ICMLA). IEEE. Orlando, FL, USA: IEEE; 2018. p. 757-62. Available from: https://doi.org/10.1109/ICMLA.2018.00120.

[7] Idrees F, Rajarajan M, Conti M, Chen TM, Rahulamathavan Y. PIndroid: A novel Android malware detection system using ensemble learning methods. Computers & Security. 2017;68:36-46. Available from: https://www.sciencedirect.com/science/article/pii/S0167404817300640.

[8] Shahriar H, Zhang C, Talukder MA, Islam S. In: Maleh Y, Shojafar M, Alazab M, Baddi Y, editors. Mobile Application Security Using Static and Dynamic Analysis. Cham: Springer International Publishing; 2021. p. 443-59. Available from: https://doi.org/10.1007/978-3-030-57024-8_20.

[9] Rajapaksha S, Senanayake J, Kalutarage H, Al-Kadri MO. AI-Powered Vulnerability Detection for Secure Source Code Development. In: Innovative Security Solutions for Information Technology and Communications. Cham: Springer Nature Switzerland; 2023. p. 275-88. Available from: https://doi.org/10.1007/978-3-031-32636-3_16.

[10] Rajapaksha S, Senanayake J, Kalutarage H, Al-Kadri MO. Enhancing Security Assurance in Software Development: AI-Based Vulnerable Code Detection with Static Analysis. In: Computer

Security. ESORICS 2023 International Workshops. Cham: Springer Nature Switzerland; 2024. p. 341-56. Available from: https://doi.org/10.1007/978-3-031-54129-2_20.

[11] Gazit T. Leveraging machine learning to find security vulnerabilities; 2024. Available from: https://github.blog/2024-02-14-fixing-security-vulnerabilities-with-ai/.

[12] Piras L, Al-Obeidallah MG, Pavlidis M, Mouratidis H, Tsohou A, Magkos E, et al. DEFeND DSM: A Data Scope Management Service for Model-Based Privacy by Design GDPR Compliance. In: Gritzalis S, Weippl ER, Kotsis G, Tjoa AM, Khalil I, editors. Trust, Privacy and Security in Digital Business. Cham: Springer International Publishing; 2020. p. 186-201. Available from: https://doi.org/10.1007/978-3-030-58986-8_13.

[13] Tsohou A, Magkos E, Mouratidis H, Chrysoloras G, Piras L, Pavlidis M, et al. Privacy, security, legal and technology acceptance elicited and consolidated requirements for a GDPR compliance platform. Information & Computer Security. 2020;28(4):531-53. Available from: https://doi.org/10.1108/ICS-01-2020-0002.

[14] Li L, Fan Y, Tse M, Lin KY. A review of applications in federated learning. Computers & Industrial Engineering. 2020;149:106854. Available from: https://doi.org/10.1016/j.cie.2020.106854.

[15] Hariharan S, Velicheti A, Anagha AS, Thomas C, Balakrishnan N. Explainable Artificial Intelligence in Cybersecurity: A Brief Review. In: 2021 4th International Conference on Security and Privacy (ISEA-ISAP); 2021. p. 1-12. Available from: https://doi.org/10.1109/ISEA-ISAP54304.2021.9689765.

[16] Senanayake J, Kalutarage H, Al-Kadri MO. Android Mobile Malware Detection Using Machine Learning: A Systematic Review. Electronics. 2021;10(13):1606. Available from: https://www.mdpi.com/2079-9292/10/13/1606.

[17] Senanayake J, Kalutarage H, Al-Kadri MO, Petrovski A, Piras L. Android Source Code Vulnerability Detection: A Systematic Literature Review. ACM Comput Surv. 2023 jan;55(9). Available from: https://doi.org/10.1145/3556974.

[18] Senanayake J, Kalutarage H, Petrovski A, Piras L, Al-Kadri MO. Defendroid: Real-time Android code vulnerability detection via blockchain federated neural network with XAI. Journal of Information Security and Applications. 2024;82:103741. Available from: https://www.sciencedirect.com/science/article/pii/S2214212624000449.

[19] Senanayake J, Kalutarage H, Al-Kadri MO, Piras L, Petrovski A. Labelled Vulnerability Dataset on Android Source Code (LVDAndro) to Develop AI-Based Code Vulnerability Detection Models. In: Proceedings of the 20th International Conference on Security and Cryptography - SECRYPT. INSTICC. SciTePress; 2023. p. 659-66. Available from: https://doi.org/10.5220/0012060400003555.

[20] Senanayake J, Kalutarage H, Al-Kadri MO, Petrovski A, Piras L. Developing Secured Android Applications by Mitigating Code Vulnerabilities with Machine Learning. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security. ASIA CCS '22. New York, NY, USA: Association for Computing Machinery; 2022. p. 1255–1257. Available from: https://doi.org/10.1145/3488932.3527290.

[21] Senanayake J, Kalutarage H, Al-Kadri MO, Petrovski A, Piras L. Android Code Vulnerabilities Early Detection Using AI-Powered ACVED Plugin. In: Atluri V, Ferrara AL, editors. Data and

Applications Security and Privacy XXXVII. Cham: Springer Nature Switzerland; 2023. p. 339-57. Available from: https://doi.org/10.1007/978-3-031-37586-6_20.

[22] Senanayake J, Kalutarage H, Petrovski A, Al-Kadri MO, Piras L. FedREVAN: Real-time DEtection of Vulnerable Android Source Code Through Federated Neural Network with XAI. In: Computer Security. ESORICS 2023 International Workshops. Cham: Springer Nature Switzerland; 2024. p. 426-41. Available from: https://doi.org/10.1007/978-3-031-54129-2_25.

[23] Senanayake J, Kalutarage H, Piras L, , Al-Kadri MO, Petrovski A. Assuring Privacy of AI-Powered Community Driven Android Code Vulnerability Detection. In: Computer Security. ESORICS 2024 International Workshops. Cham: Springer Nature Switzerland; 2024. Available from: https://www.researchgate.net/publication/384867511_Assuring_Privacy_of_AI-Powered_Community_Driven_Android_Code_Vulnerability_Detection.

[24] Delgado-Santos P, Stragapede G, Tolosana R, Guest R, Deravi F, Vera-Rodriguez R. A Survey of Privacy Vulnerabilities of Mobile Device Sensors. ACM Comput Surv. 2022 sep;54(11s). Available from: https://doi.org/10.1145/3510579.

[25] Mayrhofer R, Stoep JV, Brubaker C, Kralevich N. The Android Platform Security Model. ACM Trans Priv Secur. 2021 Apr;24(3). Available from: https://doi.org/10.1145/3448609.

[26] Sarkar A, Goyal A, Hicks D, Sarkar D, Hazra S. Android Application Development: A Brief Overview of Android Platforms and Evolution of Security Systems. In: 2019 Third International conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC). Palladam, India: IEEE; 2019. p. 73-9. Available from: https://doi.org/10.1109/I-SMAC47947.2019.9032440.

[27] Android. Platform Architecture; 2021. Accessed: 2021-08-19. Available from: https://developer.android.com/guide/platform.

[28] Senanayake J, Wijayanayake W. Applicability of crowd sourcing to determine the best transportation method by analysing user mobility. International Journal of Data Mining & Knowledge Management Process. 2018;8(4/5):27-36. Available from: https://doi.org/10.5121/ijdkp.2018.8503.

[29] Zhang J, Yao Y, Li X, Xie J, Wu G. An android vulnerability detection system. In: International Conference on Network and System Security. Springer. Cham: Springer International Publishing; 2017. p. 169-83. Available from: https://doi.org/10.1007/978-3-319-64701-2_13.

[30] Senanayake J, Pathirana N. LYZGen: A mechanism to generate leads from Generation Y and Z by analysing web and social media data. In: 2021 International Research Conference on Smart Computing and Systems Engineering (SCSE). vol. 4. Colombo, Sri Lanka: IEEE; 2021. p. 59-64. Available from: https://doi.org/10.1109/SCSE53661.2021.9568333.

[31] Gao J, Li L, Kong P, Bissyandé TF, Klein J. Understanding the Evolution of Android App Vulnerabilities. IEEE Transactions on Reliability. 2021;70(1):212-30. Available from: https://doi.org/10.1109/TR.2019.2956690.

[32] of Bristol U. Cyber Security Body of Knowledge; 2021. Accessed: 2021-10-01. Available from: https://www.cybok.org/tree/#sps.

[33] Garg S, Baliyan N. Machine Learning Based Android Vulnerability Detection: A Roadmap. In: Kanhere S, Patil VT, Sural S, Gaur MS, editors. Information Systems Security. Cham: Springer International Publishing; 2020. p. 87-93. Available from: https://doi.org/10.1007/978-3-030-65610-2_6.

[34] Linares-Vásquez M, Bavota G, Escobar-Velásquez C. An Empirical Study on Android-Related Vulnerabilities. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). Buenos Aires, Argentina: IEEE; 2017. p. 2-13. Available from: https://doi.org/10.1109/MSR.2017.60.

[35] Corporation TM. Common Vulnerability and Exposures; 2021. Accessed: 2021-10-02. Available from: https://www.cve.org/.

[36] Android. Android Security Bulletins; 2021. Accessed: 2021-10-02. Available from: https://source.android.com/security/bulletin.

[37] Nguyen DC, Wermke D, Acar Y, Backes M, Weir C, Fahl S. A Stitch in Time: Supporting Android Developers in WritingSecure Code. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. CCS '17. New York, NY, USA: Association for Computing Machinery; 2017. p. 1065–1077. Available from: https://doi.org/10.1145/3133956.3133977.

[38] Calciati P, Kuznetsov K, Gorla A, Zeller A. Automatically Granted Permissions in Android apps: An Empirical Study on their Prevalence and on the Potential Threats for Privacy. In: Proceedings of the 17th International Conference on Mining Software Repositories. Seoul Republic of Korea: Association for Computing Machinery, New York, NY, United States; 2020. p. 114-24. Available from: https://doi.org/10.1145/3379597.3387469.

[39] Garg S, Baliyan N. Comparative analysis of Android and iOS from security viewpoint. Computer Science Review. 2021;40:100372. Available from: https://doi.org/10.1016/j.cosrev.2021.100372.

[40] Malik J. Making sense of human threats and errors. Computer Fraud & Security. 2020;2020(3):6-10. Available from: https://doi.org/10.1016/S1361-3723(20)30028-2.

[41] Ponta SE, Plate H, Sabetta A, Bezzi M, Dangremont C. A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). Montreal, QC, Canada: IEEE; 2019. p. 383-7. Available from: https://doi.org/10.1109/MSR.2019.00064.

[42] Corporation M. Common Weakness Enumeration (CWE); 2023. Accessed: 2023-10-01. Available from: https://cwe.mitre.org/.

[43] of Standards NI, Technology. National Vulnerability Database; 2021. Accessed: 2021-08-21. Available from: https://nvd.nist.gov/vuln.

[44] Alzubi J, Nayyar A, Kumar A. Machine learning from theory to algorithms: an overview. In: Journal of physics: conference series. vol. 1142. IOP Publishing. Bangalore, India: IOP Publishing; 2018. p. 012012. Available from: https://doi.org/10.1088/1742-6596/1142/1/012012.

[45] Bonaccorso G. Machine learning algorithms. Birmingham, UK: Packt Publishing Ltd; 2017. Available from: https://www.packtpub.com/en-gb/product/machine-learning-algorithms-9781789347999.

[46] Srivastava G, Jhaveri RH, Bhattacharya S, Pandya S, Rajeswari, Maddikunta PKR, et al.. XAI for Cybersecurity: State of the Art, Challenges, Open Issues and Future Directions. arXiv; 2022. Available from: https://doi.org/10.48550/ARXIV.2206.03585.

[47] Babbar R, Schölkopf B. Data scarcity, robustness and extreme multi-label classification. Machine Learning. 2019;108(8-9):1329-51. Available from: https://doi.org/10.1007/s10994-019-05791-5.

[48] Alzubaidi L, Bai J, Al-Sabaawi A, Santamaría J, Albahri A, Al-dabbagh BSN, et al. A survey on deep learning tools dealing with data scarcity: definitions, challenges, solutions, tips, and applications. Journal of Big Data. 2023;10(1):46. Available from: https://doi.org/10.1186/s40537-023-00727-2.

[49] Sambasivan N, Kapania S, Highfill H, Akrong D, Paritosh P, Aroyo LM. "Everyone wants to do the model work, not the data work": Data Cascades in High-Stakes AI. In: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. CHI '21. New York, NY, USA: Association for Computing Machinery; 2021. Available from: https://doi.org/10.1145/3411764.3445518.

[50] Bansal MA, Sharma DR, Kathuria DM. A Systematic Review on Data Scarcity Problem in Deep Learning: Solution and Applications. ACM Comput Surv. 2022 sep;54(10s). Available from: https://doi.org/10.1145/3502287.

[51] Anaby-Tavor A, Carmeli B, Goldbraich E, Kantor A, Kour G, Shlomov S, et al. Do not have enough data? Deep learning to the rescue! In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 34; 2020. p. 7383-90. Available from: https://doi.org/10.1609/aaai.v34i05.6233.

[52] Li T, Sahu AK, Talwalkar A, Smith V. Federated Learning: Challenges, Methods, and Future Directions. IEEE Signal Processing Magazine. 2020;37(3):50-60. Available from: https://doi.org/10.1109/MSP.2020.2975749.

[53] Li X, Huang K, Yang W, Wang S, Zhang Z. On the Convergence of FedAvg on Non-IID Data; 2020. Available from: https://doi.org/10.48550/arXiv.1907.02189.

[54] Zakariyya I, Kalutarage H, Al-Kadri MO. Resource Efficient Federated Deep Learning for IoT Security Monitoring. In: Li W, Furnell S, Meng W, editors. Attacks and Defenses for the Internet-of-Things. Cham: Springer Nature Switzerland; 2022. p. 122-42. Available from: https://doi.org/10.1007/978-3-031-21311-3_6.

[55] Mothukuri V, Parizi RM, Pouriyeh S, Huang Y, Dehghantanha A, Srivastava G. A survey on security and privacy of federated learning. Future Generation Computer Systems. 2021;115:619-40. Available from: https://doi.org/10.1016/j.future.2020.10.007.

[56] Wei K, Li J, Ding M, Ma C, Yang HH, Farokhi F, et al. Federated Learning With Differential Privacy: Algorithms and Performance Analysis. IEEE Transactions on Information Forensics and Security. 2020;15:3454-69. Available from: https://doi.org/10.1109/TIFS.2020.2988575.

[57] Wang Y, Wang Q, Zhao L, Wang C. Differential privacy in deep learning: Privacy and beyond. Future Generation Computer Systems. 2023;148:408-24. Available from: https://doi.org/10.1016/j.future.2023.06.010.

[58] Liu B, Ding M, Shaham S, Rahayu W, Farokhi F, Lin Z. When machine learning meets privacy: A survey and outlook. ACM Computing Surveys (CSUR). 2021;54(2):1-36. Available from: https://doi.org/10.1145/3436755.

[59] Google. Tensorflow Privacy; 2024. Available from: https://github.com/tensorflow/privacy.

[60] Jiang S, Cao J, Wu H, Yang Y, Ma M, He J. BlocHIE: A BLOCkchain-Based Platform for Healthcare Information Exchange. In: 2018 IEEE International Conference on Smart Computing (SMARTCOMP); 2018. p. 49-56. Available from: https://doi.org/10.1109/SMARTCOMP.2018.00073.

[61] Zhu J, Cao J, Saxena D, Jiang S, Ferradi H. Blockchain-Empowered Federated Learning: Challenges, Solutions, and Future Directions. ACM Comput Surv. 2023 feb;55(11). Available from: https://doi.org/10.1145/3570953.

[62] Qu Y, Uddin MP, Gan C, Xiang Y, Gao L, Yearwood J. Blockchain-Enabled Federated Learning: A Survey. ACM Comput Surv. 2022 nov;55(4). Available from: https://doi.org/10.1145/3524104.

[63] Page MJ, McKenzie JE, Bossuyt PM, Boutron I, Hoffmann TC, Mulrow CD, et al. The PRISMA 2020 statement: an updated guideline for reporting systematic reviews. BMJ. 2021;372:1-9. Available from: https://doi.org/10.1136/bmj.n71.

[64] Statcounter. Mobile Operating System Market Share Worldwide; 2021. Accessed: 2022-06-02. Available from: https://gs.statcounter.com/os-market-share/mobile/worldwide/%20.

[65] Liu K, Xu S, Xu G, Zhang M, Sun D, Liu H. A Review of Android Malware Detection Approaches Based on Machine Learning. IEEE Access. 2020;8:124579-607. Available from: https://doi.org/10.1109/ACCESS.2020.3006143.

[66] Budgen D, Brereton P. Performing Systematic Literature Reviews in Software Engineering. In: Proceedings of the 28th International Conference on Software Engineering. ICSE '06. New York, NY, USA: Association for Computing Machinery; 2006. p. 1051–1052. Available from: https://doi.org/10.1145/1134285.1134500.

[67] Wohlin C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proceedings of the 18th international conference on evaluation and assessment in software engineering. New York, NY, USA: Association for Computing Machinery; 2014. p. 1-10. Available from: https://doi.org/10.1145/2601248.2601268.

[68] Kitchenham B. Procedures for performing systematic reviews. Keele, UK, Keele University. 2004;33(2004):1-26. Available from: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=29890a936639862f45cb9a987dd599dce9759bf5.

[69] Shabtai A, Fledel Y, Kanonov U, Elovici Y, Dolev S. Google Android: A State-of-the-Art Review of Security Mechanisms; 2009.

[70] Ahmed OM, Sallow AB. Android security: a review. Academic Journal of Nawroz University. 2017;6(3):135-40. Available from: https://journals.nawroz.edu.krd/index.php/ajnu/article/view/99.

[71] Li L, Bissyandé TF, Papadakis M, Rasthofer S, Bartel A, Octeau D, et al. Static analysis of android apps: A systematic literature review. Information and Software Technology. 2017;88:67-95. Available from: https://www.sciencedirect.com/science/article/pii/S0950584917302987.

[72] Kong P, Li L, Gao J, Liu K, Bissyandé TF, Klein J. Automated Testing of Android Apps: A Systematic Literature Review. IEEE Transactions on Reliability. 2019;68(1):45-66. Available from: https://doi.org/10.1109/TR.2018.2865733.

[73] Ain QU, Butt WH, Anwar MW, Azam F, Maqbool B. A Systematic Review on Code Clone Detection. IEEE Access. 2019;7:86121-44. Available from: https://doi.org/10.1109/ACCESS.2019.2918202.

[74] Garg S, Baliyan N. Android security assessment: A review, taxonomy and research gap study. Computers & Security. 2021;100:102087. Available from: https://www.sciencedirect.com/science/article/pii/S0167404820303606.

[75] Liu Y, Tantithamthavorn C, Li L, Liu Y. Deep Learning for Android Malware Defenses: A Systematic Literature Review. New York, NY, USA: Association for Computing Machinery; 2022. Just Accepted. Available from: https://doi.org/10.1145/3544968.

[76] Eberendu AC, Udegbe VI, Ezennorom EO, Ibegbulam AC, Chinebu TI, et al. A Systematic Literature Review of Software Vulnerability Detection. European Journal of Computer Science and Information Technology. 2022;10(1):23-37. Available from: https://tudr.org/id/eprint/284.

[77] Burguera I, Zurutuza U, Nadjm-Tehrani S. Crowdroid: Behavior-Based Malware Detection System for Android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices. SPSM '11. New York, NY, USA: Association for Computing Machinery; 2011. p. 15–26. Available from: https://doi.org/10.1145/2046614.2046619.

[78] Enck W, Ongtang M, McDaniel P. On Lightweight Mobile Phone Application Certification. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. CCS '09. New York, NY, USA: Association for Computing Machinery; 2009. p. 235–245. Available from: https://doi.org/10.1145/1653662.1653691.

[79] Faruki P, Ganmoor V, Laxmi V, Gaur MS, Bharmal A. AndroSimilar: Robust Statistical Feature Signature for Android Malware Detection. In: Proceedings of the 6th International Conference on Security of Information and Networks. SIN '13. New York, NY, USA: Association for Computing Machinery; 2013. p. 152–159. Available from: https://doi.org/10.1145/2523514.2523539.

[80] Grace M, Zhou Y, Zhang Q, Zou S, Jiang X. RiskRanker: Scalable and Accurate Zero-Day Android Malware Detection. In: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services. MobiSys '12. New York, NY, USA: Association for Computing Machinery; 2012. p. 281–294. Available from: https://doi.org/10.1145/2307636.2307663.

[81] Jing Y, Ahn GJ, Zhao Z, Hu H. RiskMon: Continuous and Automated Risk Assessment of Mobile Applications. In: Proceedings of the 4th ACM Conference on Data and Application Security and Privacy. CODASPY '14. New York, NY, USA: Association for Computing Machinery; 2014. p. 99–110. Available from: https://doi.org/10.1145/2557547.2557549.

[82] Russello G, Jimenez AB, Naderi H, van der Mark W. FireDroid: Hardening Security in Almost-Stock Android. In: Proceedings of the 29th Annual Computer Security Applications Conference. ACSAC '13. New York, NY, USA: Association for Computing Machinery; 2013. p. 319–328. Available from: https://doi.org/10.1145/2523649.2523678.

[83] Xu R, Saïdi H, Anderson R. Aurasium: Practical Policy Enforcement for Android Applications. In: 21st USENIX Security Symposium (USENIX Security 12). Bellevue, WA: USENIX Association; 2012. p. 539-52. Available from: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/xu_rubin.

[84] Yan LK, Yin H. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In: 21st USENIX Security Symposium (USENIX Security 12). Bellevue, WA: USENIX Association; 2012. p. 569-84. Available from: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/yan.

[85] Zhao Y, Yu T, Su T, Liu Y, Zheng W, Zhang J, et al. ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). Montreal, QC, Canada: IEEE; 2019. p. 128-39. Available from: https://doi.org/10.1109/ICSE.2019.00030.

[86] Zhou Y, Wang Z, Zhou W, Jiang X. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In: NDSS. vol. 25. Montreal, QC, Canada: NDSS; 2012. p. 50-2. Available from: https://www.cs.umd.edu/class/fall2019/cmsc818O/papers/hey-you-market.pdf.

[87] Amalfitano D, Fasolino AR, Tramontana P, De Carmine S, Memon AM. Using GUI ripping for automated testing of Android applications. In: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. Essen, Germany: IEEE; 2012. p. 258-61. Available from: https://doi.org/10.1145/2351676.2351717.

[88] Android D. UI/App Exerciser Monkey; 2022. Accessed: 2022-04-12. Available from: https://developer.android.com/studio/test/other-testing-tools/monkey.

[89] Hocke R. RaiMan's Sikulix; 2022. Accessed: 2022-04-12. Available from: http://sikulix.com/.

[90] Renas. RobotiumTech/robotium: Android UI Testing; 2016. Accessed: 2022-04-12. Available from: https://github.com/RobotiumTech/robotium.

[91] Roubtsov V. EMMA: a free Java code coverage tool; 2005. Accessed: 2022-04-12. Available from: http://emma.sourceforge.net/.

[92] Williams C. Roboelectric; 2017. Accessed: 2022-04-12. Available from: http://robolectric.org/.

[93] Lam P, Bodden E, Lhoták O, Hendren L. The Soot framework for Java program analysis: a retrospective; 2011. Accessed: 2022-04-12. Available from: http://soot-oss.github.io/soot/.

[94] Vallee-Rai R, Hendren LJ. Jimple: Simplifying Java Bytecode for Analyses and Transformations; 1998. Available from: https://keypointt.com/assets/2017_spark_spring_SF_IBM_dataframe-paper.pdf.

[95] Hur JB, Shamsi JA. A survey on security issues, vulnerabilities and attacks in Android based smartphone. In: 2017 International Conference on Information and Communication Technologies (ICICT). IEEE. Karachi, Pakistan: IEEE; 2017. p. 40-6. Available from: https://doi.org/10.1109/ICICT.2017.8320163.

[96] Alqahtani EJ, Zagrouba R, Almuhaideb A. A Survey on Android Malware Detection Techniques Using Machine Learning Algorithms. In: 2019 Sixth International Conference on Software Defined Systems (SDS). IEEE. Rome, Italy: IEEE; 2019. p. 110-7. Available from: https://doi.org/10.1109/SDS.2019.8768729.

[97] Lopes J, Serrão C, Nunes L, Almeida A, Oliveira J. Overview of machine learning methods for Android malware identification. In: 2019 7th International Symposium on Digital Forensics and Security (ISDFS). IEEE. Barcelos, Portugal: IEEE; 2019. p. 1-6. Available from: https://doi.org/10.1109/ISDFS.2019.8757523.

[98] Choudhary M, Kishore B. HAAMD: hybrid analysis for Android malware detection. In: 2018 International Conference on Computer Communication and Informatics (ICCCI). IEEE. Coimbatore, India: IEEE; 2018. p. 1-4. Available from: https://doi.org/10.1109/ICCCI.2018.8441295.

[99] Pustogarov I, Wu Q, Lie D. Ex-vivo dynamic analysis framework for Android device drivers. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE. San Francisco, CA, USA: IEEE; 2020. p. 1088-105. Available from: https://doi.org/10.1109/SP40000.2020.00094.

[100] Amin A, Eldessouki A, Magdy MT, Abdeen N, Hindy H, Hegazy I. AndroShield: automated android applications vulnerability detection, a hybrid static and dynamic analysis approach. Information. 2019;10(10):326. Available from: https://doi.org/10.3390/info10100326.

[101] Ardito L, Coppola R, Malnati G, Torchiano M. Effectiveness of Kotlin vs. Java in android app development tasks. Information and Software Technology. 2020;127:106374. Available from: https://www.sciencedirect.com/science/article/pii/S0950584920301439.

[102] Gruver B. Smali; 2022. Accessed: 2022-04-12. Available from: https://github.com/JesusFreke/smali.

[103] Pan B. dex2jar; 2022. Accessed: 2022-04-12. Available from: https://github.com/pxb1988/dex2jar.

[104] Wala. T.J. Watson Libraries for Analysis; 2022. Accessed: 2022-04-12. Available from: http://wala.sourceforge.net/.

[105] Kouliaridis V, Kambourakis G. A Comprehensive Survey on Machine Learning Techniques for Android Malware Detection. Information. 2021;12(5):185. Available from: https://doi.org/10.3390/info12050185.

[106] Li J, Sun L, Yan Q, Li Z, Srisa-An W, Ye H. Significant permission identification for machine-learning-based android malware detection. IEEE Transactions on Industrial Informatics. 2018;14(7):3216-25. Available from: https://doi.org/10.1109/TII.2017.2789219.

[107] Onwuzurike L, Mariconti E, Andriotis P, Cristofaro ED, Ross G, Stringhini G. MaMaDroid: Detecting Android malware by building Markov chains of behavioral models (extended version). ACM Transactions on Privacy and Security (TOPS). 2019;22(2):1-34. Available from: https://doi.org/10.1145/3313391.

[108] Garg S, Peddoju SK, Sarje AK. Network-based detection of Android malicious apps. International Journal of Information Security. 2017;16(4):385-400. Available from: https://doi.org/10.1007/s10207-016-0343-z.

[109] Nawaz A, et al. Feature Engineering based on Hybrid Features for Malware Detection over Android Framework. Turkish Journal of Computer and Mathematics Education (TURCOMAT). 2021;12(10):2856-64. Available from: https://www.turcomat.org/index.php/turkbilmat/article/view/4931.

[110] Tumbleson C, Wiśniewski R. Apktool V2.5.0 - A tool for reverse engineering Android apk files; 2010. Accessed: 2021-08-21. Available from: https://ibotpeaches.github.io/Apktool/.

[111] Tang J, Li R, Wang K, Gu X, Xu Z. A novel hybrid method to analyze security vulnerabilities in Android applications. Tsinghua Science and Technology. 2020;25(5):589-603. Available from: https://doi.org/10.26599/TST.2019.9010067.

[112] Wang Y, Xu G, Liu X, Mao W, Si C, Pedrycz W, et al. Identifying vulnerabilities of SSL/TLS certificate verification in Android apps with static and dynamic analysis. Journal of Systems and Software. 2020;167:110609. Available from: https://doi.org/10.1016/j.jss.2020.110609.

[113] Palomba F, Di Nucci D, Panichella A, Zaidman A, De Lucia A. Lightweight detection of android-specific code smells: The adoctor project. In: 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER). IEEE. Klagenfurt, Austria: IEEE; 2017. p. 487-91. Available from: https://doi.org/10.1109/SANER.2017.7884659.

[114] Sun Y, Xie Y, Qiu Z, Pan Y, Weng J, Guo S. Detecting Android malware based on extreme learning machine. In: 2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on

Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech). IEEE. Orlando, FL, USA: IEEE; 2017. p. 47-53. Available from: https://doi.org/10.1109/DASC-PICom-DataCom-CyberSciTec.2017.24.

[115] Gajrani J, Tripathi M, Laxmi V, Somani G, Zemmari A, Gaur MS. Vulvet: Vetting of Vulnerabilities in Android Apps to Thwart Exploitation. Digital Threats: Research and Practice. 2020 May;1(2). Available from: https://doi.org/10.1145/3376121.

[116] Yang X, Lo D, Li L, Xia X, Bissyandé TF, Klein J. Characterizing malicious Android apps by mining topic-specific data flow signatures. Information and Software Technology. 2017;90:27-39. Available from: https://www.sciencedirect.com/science/article/pii/S095058491730366X.

[117] Kim S, Yeom S, Oh H, Shin D, Shin D. Automatic Malicious Code Classification System through Static Analysis Using Machine Learning. Symmetry. 2021;13(1):35. Available from: https://doi.org/10.3390/sym13010035.

[118] Bilgin Z, Ersoy MA, Soykan EU, Tomur E, Çomak P, Karaçay L. Vulnerability Prediction From Source Code Using Machine Learning. IEEE Access. 2020;8:150672-84. Available from: https://doi.org/10.1109/ACCESS.2020.3016774.

[119] of Standards NI, Technology. Software Assurance Metrics And Tool Evaluation (SAMATE); 2021. Accessed: 2021-08-21. Available from: https://www.nist.gov/itl/ssd/software-quality-group/samate.

[120] of Standards NI, Technology. Static Analysis Tool Exposition (SATE) IV; 2021. Accessed: 2021-08-21. Available from: https://www.nist.gov/itl/ssd/software-quality-group/static-analysis-tool-exposition-sate-iv.

[121] Chernis B, Verma R. Machine learning methods for software vulnerability detection. In: Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics. New York, NY, USA: Association for Computing Machinery; 2018. p. 31-9. Available from: https://doi.org/10.1145/3180445.3180453.

[122] Samples V. Github Malware Dataset; 2020. Accessed: 2021-09-19. Available from: https://github.com/topics/malware-dataset.

[123] Arp D, Spreitzenbarth M, Hubner M, Gascon H, Rieck K, Siemens C. Drebin: Effective and explainable detection of android malware in your pocket. In: Ndss. vol. 14. San Diego, CA, USA: NDSS; 2014. p. 23-6. Available from: https://prosec.mlsec.org/docs/2014-ndss.pdf.

[124] Mahindru A, Singh P. Dynamic permissions based android malware detection using machine learning techniques. In: Proceedings of the 10th innovations in software engineering conference. New York, NY, USA: Association for Computing Machinery; 2017. p. 202-10. Available from: https://doi.org/10.1145/3021460.3021485.

[125] Wu F, Wang J, Liu J, Wang W. Vulnerability detection with deep learning. In: 2017 3rd IEEE International Conference on Computer and Communications (ICCC). IEEE. Chengdu, China: IEEE; 2017. p. 1298-302. Available from: https://doi.org/10.1109/CompComm.2017.8322752.

[126] Pang Y, Xue X, Wang H. Predicting vulnerable software components through deep neural network. In: Proceedings of the 2017 International Conference on Deep Learning Technologies. New York, NY, USA: Association for Computing Machinery; 2017. p. 6-10. Available from: https://doi.org/10.1145/3094243.3094245.

[127] Gupta A, Suri B, Kumar V, Jain P. Extracting rules for vulnerabilities detection with static metrics using machine learning. International Journal of System Assurance Engineering and Management. 2021;12(1):65-76. Available from: https://doi.org/10.1007/s13198-020-01036-0.

[128] Sikder AK, Aksu H, Uluagac AS. 6thsense: A context-aware sensor-based attack detector for smart devices. In: 26th {USENIX} Security Symposium ({USENIX} Security 17). Vancouver, BC: USENIX Association; 2017. p. 397-414. Available from: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/sikder.

[129] Malik Y, Campos CRS, Jaafar F. Detecting Android Security Vulnerabilities Using Machine Learning and System Calls Analysis. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C). Sofia, Bulgaria: IEEE; 2019. p. 109-13. Available from: https://doi.org/10.1109/QRS-C.2019.00033.

[130] Zhuo L, Zhimin G, Cen C. Research on Android intent security detection based on machine learning. In: 2017 4th International Conference on Information Science and Control Engineering (ICISCE). IEEE. Changsha, China: IEEE; 2017. p. 569-74. Available from: https://doi.org/10.1109/ICISCE.2017.124.

[131] Garg S, Baliyan N. A novel parallel classifier scheme for vulnerability detection in Android. Computers & Electrical Engineering. 2019;77:12-26. Available from: https://www.sciencedirect.com/science/article/pii/S0045790618320123.

[132] Jannat US, Hasnayeen SM, Shuhan MKB, Ferdous MS. Analysis and detection of malware in Android applications using machine learning. In: 2019 International Conference on Electrical, Computer and Communication Engineering (ECCE). IEEE. Cox'sBazar, Bangladesh: IEEE; 2019. p. 1-7. Available from: https://doi.org/10.1109/ECACE.2019.8679493.

[133] Foundation PS. Androguard; 2019. Accessed: 2021-09-19. Available from: https://pypi.org/project/androguard/.

[134] Kaggle. Google Playstore Appsin Kaggle; 2020. Accessed: 2021-09-19. Available from: https://www.kaggle.com/gauthamp10/google-playstore-apps.

[135] Zhou Y, Jiang X. Dissecting android malware: Characterization and evolution. In: 2012 IEEE symposium on security and privacy. IEEE. San Francisco, CA, USA: IEEE; 2012. p. 95-109. Available from: https://doi.org/10.1109/SP.2012.16.

[136] Leeds M, Keffeler M, Atkison T. A comparison of features for android malware detection. In: Proceedings of the SouthEast Conference. New York, NY, USA: Association for Computing Machinery; 2017. p. 63-8. Available from: https://doi.org/10.1145/3077286.3077288.

[137] Surendran R, Thomas T, Emmanuel S. A TAN based hybrid model for android malware detection. Journal of Information Security and Applications. 2020;54:102483. Available from: https://doi.org/10.1016/j.jisa.2020.102483.

[138] Alsayra. Intelligence and Security Informatics Data Sets; 2020. Accessed: 2021-09-19. Available from: https://www.azsecure-data.org.

[139] Lab A. Android Malware Dataset; 2020. Accessed: 2021-09-19. Available from: http://amd.arguslab.org/.

[140] Microsoft. Microsoft Malware Classification Challenge (Big 2015); 2022. Accessed: 2022-04-12. Available from: https://www.kaggle.com/c/malware-classification/.

[141] Alon U, Zilberstein M, Levy O, Yahav E. code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages. 2019;3(POPL):1-29. Available from: https://doi.org/10.1145/3290353.

[142] Grieco G, Grinblat GL, Uzal L, Rawat S, Feist J, Mounier L. Toward Large-Scale Vulnerability Discovery Using Machine Learning. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. CODASPY '16. New York, NY, USA: Association for Computing Machinery; 2016. p. 85–96. Available from: https://doi.org/10.1145/2857705.2857720.

[143] Gultnieks C. F-Droid - Free and Open Source Android App Repository; 2022. Accessed: 2022-04-12. Available from: https://f-droid.org/.

[144] Tarasevich S. Android Universal Image Loader; 2022. Accessed: 2022-04-12. Available from: https://github.com/nostra13/Android-Universal-Image-Loader.

[145] Gamma E. JHotDraw; 2022. Accessed: 2022-04-12. Available from: https://sourceforge.net/projects/jhotdraw/.

[146] Inc G. Google Play; 2020. Accessed: 2021-09-19. Available from: https://play.google.com/.

[147] Technology BZ. Wandoujia App Market; 2020. Accessed: 2021-09-19. Available from: https://www.wandoujia.com/apps/.

[148] Allix K, Bissyandé TF, Klein J, Le Traon Y. AndroZoo: Collecting Millions of Android Apps for the Research Community. In: Proceedings of the 13th International Conference on Mining Software Repositories. MSR '16. New York, NY, USA: ACM; 2016. p. 468-71. Available from: http://doi.acm.org/10.1145/2901739.2903508.

[149] Bagheri H, Kang E, Malek S, Jackson D. A formal approach for detection of security flaws in the android permission system. Formal Aspects of Computing. 2018;30(5):525-44. Available from: https://doi.org/10.1007/s00165-017-0445-z.

[150] Chang Y, Liu B, Cong L, Deng H, Li J, Chen Y. Vulnerability Parser: A Static Vulnerability Analysis System for Android Applications. Journal of Physics: Conference Series. 2019 aug;1288:012053. Available from: https://doi.org/10.1088/1742-6596/1288/1/012053.

[151] Zhan X, Fan L, Chen S, We F, Liu T, Luo X, et al. ATVHunter: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). Madrid, ES: IEEE; 2021. p. 1695-707. Available from: https://doi.org/10.1109/ICSE43902.2021.00150.

[152] El-Zawawy MA, Losiouk E, Conti M. Vulnerabilities in Android webview objects: Still not the end! Computers & Security. 2021;109:102395. Available from: https://www.sciencedirect.com/science/article/pii/S0167404821002194.

[153] Li L, Bissyandé TF, Octeau D, Klein J. DroidRA: Taming Reflection to Support Whole-Program Analysis of Android Apps. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ISSTA 2016. New York, NY, USA: Association for Computing Machinery; 2016. p. 318–329. Available from: https://doi.org/10.1145/2931037.2931044.

[154] Qin J, Zhang H, Guo J, Wang S, Wen Q, Shi Y. Vulnerability Detection on Android Apps–Inspired by Case Study on Vulnerability Related With Web Functions. IEEE Access. 2020;8:106437-51. Available from: https://doi.org/10.1109/ACCESS.2020.2998043.

[155] Center NCNETC. China National Vulnerability Database; 2021. Accessed: 2021-10-02. Available from: https://www.cnvd.org.cn/.

[156] Corporation TM. Common Weakness Enumeration; 2021. Accessed: 2021-10-02. Available from: https://cwe.mitre.org/.

[157] Shezan FH, Afroze SF, Iqbal A. Vulnerability detection in recent Android apps: An empirical study. In: 2017 International Conference on Networking, Systems and Security (NSysS). Dhaka, Bangladesh: IEEE; 2017. p. 55-63. Available from: https://doi.org/10.1109/NSysS.2017.7885802.

[158] (EATL) EATL. EATL App Store; 2021. Accessed: 2021-10-02. Available from: http://eatlapps.com/apps/store.

[159] Lin YC. AndroBugs; 2015. Accessed: 2021-08-21. Available from: https://github.com/AndroBugs/.

[160] of Xi'an Jiaotong University BRT. SandDroid - An automatic Android application analysis system; 2022. Accessed: 2022-04-12. Available from: http://sanddroid.xjtu.edu.cn/.

[161] LinkedIn. Quick Android Review Kit (QARK); 2015. Accessed: 2021-08-21. Available from: https://github.com/linkedin/qark/.

[162] Chao W, Qun L, XiaoHu W, TianYu R, JiaHan D, GuangXin G, et al. An Android Application Vulnerability Mining Method Based On Static and Dynamic Analysis. In: 2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC). Chongqing, China: IEEE; 2020. p. 599-603. Available from: https://doi.org/10.1109/ITOEC49072.2020.9141575.

[163] Android. Apkanalyzer; 2022. Accessed: 2022-04-12. Available from: https://developer.android.com/studio/command-line/apkanalyzer.

[164] Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, et al. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '14. New York, NY, USA: Association for Computing Machinery; 2014. p. 259–269. Available from: https://doi.org/10.1145/2594291.2594299.

[165] Security. Qihoo; 2022. Accessed: 2022-04-12. Available from: http://zhushou.360.cn/.

[166] Jackson D. Software Abstractions: logic, language, and analysis. Cambridge, Massachusetts, London, England: MIT press; 2012. Available from: https://mitpress.mit.edu/9780262528900/software-abstractions/.

[167] Rizzo C, Cavallaro L, Kinder J. Babelview: Evaluating the impact of code injection attacks in mobile webviews. In: International Symposium on Research in Attacks, Intrusions, and Defenses. Springer. Cham: Springer International Publishing; 2018. p. 25-46. Available from: https://doi.org/10.1007/978-3-030-00470-5_2.

[168] Huawei. AppGallery; 2022. Accessed: 2022-04-12. Available from: https://appgallery.huawei.com/Featured.

[169] Tahaei M, Vaniea K, Beznosov K, Wolters MK. Security Notifications in Static Analysis Tools: Developers' Attitudes, Comprehension, and Ability to Act on Them. In: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. New York, NY, USA: Association for Computing Machinery; 2021. p. 1-17. Available from: https://doi.org/10.1145/3411764.3445616.

[170] Goaër OL. Enforcing green code with Android lint. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops. New York, NY, USA: Association for Computing Machinery; 2020. p. 85-90. Available from: https://doi.org/10.1145/3417113.3422188.

[171] Habchi S, Blanc X, Rouvoy R. On adopting linters to deal with performance concerns in android apps. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE. Montpellier, France: IEEE; 2018. p. 6-16. Available from: https://doi.org/10.1145/3238147.3238197.

[172] Wei L, Liu Y, Cheung SC. OASIS: prioritizing static analysis warnings for Android apps based on app user reviews. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. New York, NY, USA: Association for Computing Machinery; 2017. p. 672-82. Available from: https://doi.org/10.1145/3106237.3106294.

[173] Luo L, Dolby J, Bodden E. MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper). In: 33rd European Conference on Object-Oriented Programming (ECOOP 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik; 2019. p. 21:1-21:25. Available from: http://drops.dagstuhl.de/opus/volltexte/2019/10813.

[174] Labs X. Devknox - Security plugin for Android Studio; 2016. Accessed: 2021-08-21. Available from: https://devknox.io/.

[175] Rahman MS, Kojusner B, Kennedy R, Pathak P, Qi L, Williams B. So U R CERER: Developer-Driven Security Testing Framework for Android Apps. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW). Melbourne, Australia: IEEE; 2021. p. 40-6. Available from: https://doi.org/10.1109/ASEW52652.2021.00020.

[176] Ma S, Thung F, Lo D, Sun C, Deng RH. Vurle: Automatic vulnerability detection and repair by learning from examples. In: European Symposium on Research in Computer Security. Springer. Cham: Springer International Publishing; 2017. p. 229-46. Available from: https://doi.org/10.1007/978-3-319-66399-9_13.

[177] Miller T. Explanation in artificial intelligence: Insights from the social sciences. Artificial Intelligence. 2019;267:1-38. Available from: https://www.sciencedirect.com/science/article/pii/S0004370218305988.

[178] Nguyen TN, Choo R. Human-in-the-Loop XAI-enabled Vulnerability Detection, Investigation, and Mitigation. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE); 2021. p. 1210-2. Available from: https://doi.org/10.1109/ASE51524.2021.9678840.

[179] Wijekoon A, Wiratunga N. A user-centred evaluation of DisCERN: Discovering counterfactuals for code vulnerability detection and correction. Knowledge-Based Systems. 2023;278:110830. Available from: https://www.sciencedirect.com/science/article/pii/S0950705123005804.

[180] Bhatnagar P. Explainable AI (XAI) — A guide to 7 Packages in Python to Explain Your Models; 2021. Accessed: 2023-03-20. Available from: https://towardsdatascience.com/explainable-ai-xai-a-guide-to-7-packages_in-python-to-explain-your-models-932967f0634b.

[181] Marchetto A. Can explainability and deep-learning be used for localizing vulnerabilities in source code? In: Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024). AST '24. New York, NY, USA: Association for Computing Machinery; 2024. p. 110–119. Available from: https://doi.org/10.1145/3644032.3644448.

[182] Li D, Liu Y, Huang J. Assessment of Software Vulnerability Contributing Factors by Model-Agnostic Explainable AI. Machine Learning and Knowledge Extraction. 2024;6(2):1087-113. Available from: https://www.mdpi.com/2504-4990/6/2/50.

[183] Weir C, Becker I, Noble J, Blair L, Sasse MA, Rashid A. Interventions for long-term software security: Creating a lightweight program of assurance techniques for developers. Software: Practice and Experience. 2020;50(3):275-98. Available from: https://doi.org/10.1002/spe.2774.

[184] Ranganath VP, Mitra J. Are free android app security analysis tools effective in detecting known vulnerabilities? Empirical Software Engineering. 2020;25(1):178-219. Available from: https://doi.org/10.1007/s10664-019-09749-y.

[185] Mitra J, Ranganath VP. Ghera: A Repository of Android App Vulnerability Benchmarks. In: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering. PROMISE. New York, NY, USA: Association for Computing Machinery; 2017. p. 43–52. Available from: https://doi.org/10.1145/3127005.3127010.

[186] Alenezi M, Almomani I. Empirical analysis of static code metrics for predicting risk scores in android applications. In: 5th International Symposium on Data Mining Applications. Springer. Cham: Springer International Publishing; 2018. p. 84-94. Available from: https://doi.org/10.1007/978-3-319-78753-4_8.

[187] Patil M, Pramod D. AndRev: Reverse Engineering Tool to Extract Permissions of Android Mobile Apps for Analysis. In: Computer Networks and Inventive Communication Technologies. Singapore: Springer Singapore; 2021. p. 1199-207. Available from: https://doi.org/10.1007/978-981-15-9647-6_95.

[188] Mcdonald J, Herron N, Glisson W, Benton R. Machine Learning-Based Android Malware Detection Using Manifest Permissions. In: Proceedings of the 54th Hawaii International Conference on System Sciences. USA: HICSS; 2021. p. 6976. Available from: https://doi.org/10.24251/HICSS.2021.839.

[189] Fahl S, Harbach M, Muders T, Baumgärtner L, Freisleben B, Smith M. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. CCS '12. New York, NY, USA: Association for Computing Machinery; 2012. p. 50–61. Available from: https://doi.org/10.1145/2382196.2382205.

[190] Bagheri H, Sadeghi A, Garcia J, Malek S. COVERT: Compositional Analysis of Android Inter-App Permission Leakage. IEEE Transactions on Software Engineering. 2015;41(9):866-86. Available from: https://doi.org/10.1109/TSE.2015.2419611.

[191] Calzavara S, Grishchenko I, Maffei M. HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving. In: 2016 IEEE European Symposium on Security and Privacy (EuroS P). Saarbruecken, Germany: IEEE; 2016. p. 47-62. Available from: https://doi.org/10.1109/EuroSP.2016.16.

[192] Flankerhqd. JAADAS: Joint Advanced Application Defect Assessment for Android Application; 2016. Accessed: 2021-08-21. Available from: https://github.com/flankerhqd/JAADAS.

[193] Bosu A, Liu F, Yao DD, Wang G. Collusive Data Leak and More: Large-Scale Threat Analysis of Inter-App Communications. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. ASIA CCS '17. New York, NY, USA: Association for Computing Machinery; 2017. p. 71–85. Available from: https://doi.org/10.1145/3052973.3053004.

[194] Kalutarage HK, Nguyen HN, Shaikh SA. Towards a threat assessment framework for apps collusion. Telecommunication systems. 2017;66(3):417-30. Available from: https://doi.org/10.1007/s11235-017-0296-1.

[195] Lindorfer M, Neugschwandtner M, Platzer C. MARVIN: Efficient and Comprehensive Mobile App Classification through Static and Dynamic Analysis. In: 2015 IEEE 39th Annual Computer Software and Applications Conference. vol. 2. Taichung, Taiwan: IEEE; 2015. p. 422-33. Available from: https://doi.org/10.1109/COMPSAC.2015.103.

[196] OpenSecurity. Mobile Security Framework (MobSF); 2015. Accessed: 2021-08-21. Available from: https://github.com/MobSF/Mobile-Security-Framework-MobSF.

[197] Wei F, Roy S, Ou X, Robby. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. CCS '14. New York, NY, USA: Association for Computing Machinery; 2014. p. 1329–1341. Available from: https://doi.org/10.1145/2660267.2660357.

[198] Snyk. Snyk Developer Security; 2023. Accessed: 2023-10-01. Available from: https://snyk.io/.

[199] immuniweb. Mobile Application Penetration Testing | ImmuniWeb Mobile Suite; 2023. Accessed: 2023-10-01. Available from: https://www.immuniweb.com/products/mobile/.

[200] WithSecure. The leading security testing framework for Android; 2023. Accessed: 2023-10-01. Available from: https://labs.withsecure.com/tools/drozer.

[201] Sharma S, Krishna A. Hacker Style Pentest by Astra Security; 2023. Accessed: 2023-10-01. Available from: https://www.getastra.com/pentest.

[202] Technologies T. AppChina; 2021. Accessed: 2021-09-19. Available from: https://tracxn.com/d/companies/appchina.com/.

[203] Tencent. Tencent PC Manager; 2020. Accessed: 2021-09-19. Available from: https://www.pcmgr-global.com/.

[204] System STC. YingYongBao; 2018. Accessed: 2021-09-19. Available from: https://android.myapp.com/.

[205] Trust IC. Contagio; 2020. Accessed: 2021-09-19. Available from: https://www.impactcybertrust.org/dataset_view?idDataset=1273.

[206] VirusShare. VirusShare - Because Sharing is Caring; 2020. Accessed: 2021-09-19. Available from: https://virusshare.com/.

[207] Steppa. Intel Security/ MacAfee; 2020. Accessed: 2021-09-19. Available from: https://steppa.ca/portfolio-view/malware-threat-intel-datasets/.

[208] Chen K, Wang P, Lee Y, Wang X, Zhang N, Huang H, et al. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In: 24th {USENIX} Security Symposium ({USENIX} Security 15). Washington, D.C.: USENIX Association; 2015. p. 659-74. Available from: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/chen-kai.

[209] Team A. APKPure; 2020. Accessed: 2021-09-19. Available from: https://m.apkpure.com/.

[210] Data M. Anrdoid Permission Dataset; 2020. Accessed: 2021-09-19. Available from: https://data.mendeley.com/datasets/b4mxg7ydb7/3.

[211] Maggi F, Valdi A, Zanero S. Andrototal: A flexible, scalable toolbox and service for testing mobile malware detectors. In: Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices. New York, NY, USA: Association for Computing Machinery; 2013. p. 49-54. Available from: https://doi.org/10.1145/2516760.2516768.

[212] of New Brunswick U. CICMaldroid Dataset; 2020. Accessed: 2021-09-19. Available from: https://www.unb.ca/cic/datasets/maldroid-2020.html.

[213] Ponta SE, Plate H, Sabetta A, Bezzi M, Dangremont C. A manually-curated dataset of fixes to vulnerabilities of open-source software. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). IEEE. Montreal, QC, Canada: IEEE; 2019. p. 383-7. Available from: https://doi.org/10.1109/MSR.2019.00064.

[214] Namrud Z, Kpodjedo S, Talhi C. AndroVul: a repository for Android security vulnerabilities. In: Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering. USA: IBM Corp.; 2019. p. 64-71. Available from: https://dl.acm.org/doi/abs/10.5555/3370272.3370279.

[215] Allix K, Bissyandé TF, Klein J, Le Traon Y. AndroZoo: Collecting Millions of Android Apps for the Research Community. In: Proceedings of the 13th International Conference on Mining Software Repositories. MSR '16. New York, NY, USA: ACM; 2016. p. 468–471. Available from: https://doi.org/10.1145/2901739.2903508.

[216] Challande A, David R, Renault G. Building a Commit-level Dataset of Real-world Vulnerabilities. In: Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy; 2022. p. 101-6. Available from: https://doi.org/10.1145/3508398.3511495.

[217] Karmaker ("Santu") SK, Hassan MM, Smith MJ, Xu L, Zhai C, Veeramachaneni K. AutoML to Date and Beyond: Challenges and Opportunities. ACM Comput Surv. 2021 oct;54(8). Available from: https://doi.org/10.1145/3470918.

[218] Simonin D. Fossdroid; 2022. Accessed: 2023-01-02. Available from: https://fossdroid.com/.

[219] Hanif H, Maffeis S. VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. In: 2022 International Joint Conference on Neural Networks (IJCNN); 2022. p. 1-8. Available from: https://doi.org/10.1109/IJCNN55064.2022.9892280.

[220] Mazuera-Rozo A, Escobar-Velásquez C, Espitia-Acero J, Vega-Guzmán D, Trubiani C, Linares-Vásquez M, et al. Taxonomy of security weaknesses in Java and Kotlin Android apps. Journal of Systems and Software. 2022;187:111233. Available from: https://www.sciencedirect.com/science/article/pii/S0164121222000103.

[221] Feurer M, Klein A, Eggensperger K, Springenberg J, Blum M, Hutter F. Efficient and Robust Automated Machine Learning. In: Advances in Neural Information Processing Systems 28 (2015); 2015. p. 2962-70. Available from: https://proceedings.neurips.cc/paper_files/paper/2015/file/11d0e6287202fced83f79975ec59a3a6-Paper.pdf.

[222] Feurer M, Eggensperger K, Falkner S, Lindauer M, Hutter F. Auto-Sklearn 2.0: Hands-free AutoML via Meta-Learning. arXiv:200704074 [csLG]. 2020. Available from: http://jmlr.org/papers/v23/21-0992.html.

[223] scikit-learn developers. StackingClassifier; 2022. Available from: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.StackingClassifier.html.

[224] Alexandropoulos SAN, Aridas CK, Kotsiantis SB, Vrahatis MN. Stacking Strong Ensembles of Classifiers. In: MacIntyre J, Maglogiannis I, Iliadis L, Pimenidis E, editors. Artificial Intelligence Applications and Innovations. Cham: Springer International Publishing; 2019. p. 545-56. Available from: https://doi.org/10.1007/978-3-030-19823-7_46.

[225] Google. TensorFlow Model Optimization; 2023. Accessed: 2023-04-02. Available from: https://www.tensorflow.org/model_optimization.

[226] Piras L, Al-Obeidallah MG, Praitano A, Tsohou A, Mouratidis H, Gallego-Nicasio Crespo B, et al. DEFeND Architecture: A Privacy by Design Platform for GDPR Compliance. In: Gritzalis S, Weippl ER, Katsikas SK, Anderst-Kotsis G, Tjoa AM, Khalil I, editors. Trust, Privacy and Security in Digital Business. Cham: Springer International Publishing; 2019. p. 78-93. Available from: https://doi.org/10.1007/978-3-030-27813-7_6.

[227] Beutel DJ, Topal T, Mathur A, Qiu X, Fernandez-Marques J, Gao Y, et al.. Flower: A Friendly Federated Learning Research Framework; 2022. Available from: https://doi.org/10.48550/arXiv.2007.14390.

[228] Vujicic D, Jagodic D, Randic S. Blockchain technology, bitcoin, and Ethereum: A brief overview. In: 2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH); 2018. p. 1-6. Available from: https://doi.org/10.1109/INFOTEH.2018.8345547.

[229] Androulaki E, Barger A, Bortnikov V, Cachin C, Christidis K, De Caro A, et al. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In: Proceedings of the Thirteenth EuroSys Conference. EuroSys '18. New York, NY, USA: Association for Computing Machinery; 2018. p. 1-15. Available from: https://doi.org/10.1145/3190508.3190538.

[230] Sharma DK, Pant S, Sharma M, Brahmachari S. Chapter 13 - Cryptocurrency Mechanisms for Blockchains: Models, Characteristics, Challenges, and Applications. In: Krishnan S, Balas VE, Julie EG, Robinson YH, Balaji S, Kumar R, editors. Handbook of Research on Blockchain Technology. Academic Press; 2020. p. 323-48. Available from: https://www.sciencedirect.com/science/article/pii/B9780128198162000137.

[231] Abadi M, Chu A, Goodfellow I, McMahan HB, Mironov I, Talwar K, et al. Deep Learning with Differential Privacy. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. CCS '16. New York, NY, USA: Association for Computing Machinery; 2016. p. 308–318. Available from: https://doi.org/10.1145/2976749.2978318.

[232] Google. Differential Privacy; 2023. Available from: https://github.com/google/differential-privacy.

[233] Andrew G, Thakkar O, McMahan B, Ramaswamy S. Differentially private learning with adaptive clipping. Advances in Neural Information Processing Systems. 2021;34:17455-66. Available from: https://proceedings.neurips.cc/paper_files/paper/2021/file/91cff01af640a24e7f9f7a5ab407889f-Paper.pdf.

[234] Abraham A, Magaofei, Dobrushin M, Nadal V. Mobile Security Framework (MobSF); 2023. Accessed: 2023-03-11. Available from: https://github.com/MobSF/Mobile-Security-Framework-MobSF.

[235] Acar A, Aksu H, Uluagac AS, Conti M. A Survey on Homomorphic Encryption Schemes: Theory and Implementation. ACM Comput Surv. 2018 jul;51(4). Available from: https://doi.org/10.1145/3214303.

[236] Lindell Y. Secure multiparty computation. Commun ACM. 2020 dec;64(1):86–96. Available from: https://doi.org/10.1145/3387108.

Appendix A

Survey on Identifying the Characteristics of Selected Android Developers

The survey includes the following questions to determine the traits of the chosen Android developers. The questions are labelled with 'Ch' for characteristic-related queries and answers with 'A'.

Ch-Q1: What is your age group?

A1: Below 18 years

A2: 18-25 years

A3: 25-35 years

A4: 35-45 years

A5: 45-55 years

A6: Above 55 years

Ch-Q2: What is your highest education level?

A1: Research Degree

A2: Masters Degree

A3: Bachelors Degree

A4: Higher Diploma

A5: Diploma

A6: Certificate

Ch-Q3: What is your designation?

A1: Software Engineer

A2: Senior Software Engineer

A3: Tech Lead

A4: Senior Tech Lead

A5: Software Architect

A6: Manager

Ch-Q4: What type of company are you working for?

A1: Freelancing

A2: Startup

A3: Software development firm with less than 100 developers

A4: Software development firm with 100-500 developers

A5: Software development form with more than 500 developers

A6: Software development department of a non-IT-based company

Ch-Q5: How many years of experience do you have in Software development?

A1: Less than 1 years

A2: 1 to 3 years

A3: 3 to 5 years

A4: 5 to 7 years

A5: 7 to 10 years

A6: More than 10 years

Ch-Q6: How many years of experience do you have in Android app development?

A1: Less than 1 years

A2: 1 to 3 years

A3: 3 to 5 years

A4: 5 to 7 years

A5: 7 to 10 years

A6: More than 10 years

Ch-Q7: How can you rate yourself for the level of knowledge of Android?

A1: Beginner

A2: Intermediate

A3: Advanced

Ch-Q8: Have you taken any software security training courses?

A1: Yes

A2: No

Ch-Q9: How can you rate yourself for the level of knowledge of secure coding?

A1: Beginner

A2: Intermediate

A3: Advanced

Ch-Q10: How can you rate yourself for the level of knowledge on software vulnerabilities and bugs?

A1: Beginner

A2: Intermediate

A3: Advanced

Appendix B

Developer Feedback on the Plugin

The comments that were received from the developers have been classified according to the plugin's capabilities and recommendations for enhancements.

- Capabilities:
  - This plugin tackles a genuine challenge encountered during app development, aiming to reduce code vulnerabilities.
  - In my view, this represents an excellent approach for developing secure Android apps.
  - An impressive tool with significant potential.
  - The prediction time is remarkable, and it aligns seamlessly with my usual development process.
  - This plugin is fantastic! I no longer need to consult my ethical hacking team to identify security issues.
  - The plugin delivers highly accurate results.
  - The recommendations and prediction probabilities offer robust support for addressing vulnerabilities.
  - This plugin is excellent. Consider making it official with Android Studio's endorsement.

  - Impressive work that simplifies our lives.
  - This is truly remarkable. Expanding its functionality to other programming languages would be beneficial.
  - A very valuable tool for developers dealing with security vulnerabilities during app development, providing real-time assistance to mitigate risks.
  - A commendable approach to ensuring Android app security, seamlessly integrated into the development process with minimal disruption.
  - Instantly identify vulnerabilities as you write code, streamlining the debugging process.
  - A quick, accurate, and reliable tool for detecting Android code vulnerabilities.
  - A highly valuable tool, and the fact that it is free is appreciated. Hopefully, it remains free in the future as well.

- Suggestions for Improvement:
  - It would be beneficial if the plugin could identify vulnerabilities as soon as I type a code line, without waiting for key combinations.
  - Starting the backend model as soon as Android Studio opens would be an improvement.
  - While prediction probability values are helpful, additional guidance—such as suggesting vulnerability-free code along with probabilities—would enhance usability.
  - Improving the appearance of plugin notifications from a UI perspective would be valuable.
  - The plugin's capabilities are impressive, but usability refinements are needed.
  - Extending its vulnerability detection to cover more types would be highly beneficial.
  - The "Quick Check" and "Detail Check" options are fantastic. Consider adding a feature to scan an entire project at once for a comprehensive experience.