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* [online], 82, article number 103741. Available from: https://doi.org/10.1016/j.jisa.2024.103741

Defendroid: real-time Android code vulnerability detection via blockchain federated neural network with XAI.

SENANAYAKE, J., KALUTARAGE, H., PETROVSKI, A., PIRAS, L. and AL-KADRI, M.O.

2024

OpenAIR @RGU

This document was downloaded from https://openair.rgu.ac.uk

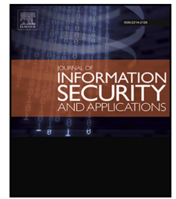
CC BY

Journal of Information Security and Applications 82 (2024) 103741

Contents lists available at ScienceDirect

Journal of Information Security and Applications

journal homepage: www.elsevier.com/locate/jisa



Defendroid: Real-time Android code vulnerability detection via blockchain federated neural network with XAI

Janaka Senanayake [a,b,*], Harsha Kalutarage [a], Andrei Petrovski [a], Luca Piras [c], Mhd Omar Al-Kadri [d]

[a] School of Computing, Robert Gordon University, Aberdeen, AB10 7QB, United Kingdom
[b] University of Kelaniya, Kelaniya, Sri Lanka
[c] Department of Computer Science, Middlesex University, London, NW4 4BT, United Kingdom
[d] University of Doha for Science and Technology, Doha, Qatar

ARTICLE INFO

Keywords:
Android application protection
Code vulnerability
Neural network
Federated learning
Source code privacy
Explainable AI
Blockchain

ABSTRACT

Ensuring strict adherence to security during the phases of Android app development is essential, primarily due to the prevalent issue of apps being released without adequate security measures in place. While a few automated tools are employed to reduce potential vulnerabilities during development, their effectiveness in detecting vulnerabilities may fall short. To address this, "Defendroid", a blockchain-based federated neural network enhanced with Explainable Artificial Intelligence (XAI) is introduced in this work. Trained on the LVDAndro dataset, the vanilla neural network model achieves a 96% accuracy and 0.96 F1-Score in binary classification for vulnerability detection. Additionally, in multi-class classification, the model accurately identifies Common Weakness Enumeration (CWE) categories with a 93% accuracy and 0.91 F1-Score. In a move to foster collaboration and model improvement, the model has been deployed within a blockchain-based federated environment. This environment enables community-driven collaborative training and enhancements in partnership with other clients. The extended model demonstrates improved accuracy of 96% and F1-Score of 0.96 in both binary and multi-class classifications. The use of XAI plays a pivotal role in presenting vulnerability detection results to developers, offering prediction probabilities for each word within the code. This model has been integrated into an Application Programming Interface (API) as the backend and further incorporated into Android Studio as a plugin, facilitating real-time vulnerability detection. Notably, Defendroid exhibits high efficiency, delivering prediction probabilities for a single code line in an average processing time of a mere 300 ms. The weight-sharing transparency in the blockchain-driven federated model enhances trust and traceability, fostering community engagement while preserving source code privacy and contributing to accuracy improvement.

1. Introduction

Ensuring the timely detection and remediation of source code vulnerabilities is of paramount importance in the secure development of Android applications. Specifically, commencing this critical process during the early phases of application development is crucial, as it significantly reduces the potential for attackers to discover and exploit vulnerabilities [1]. Android, which currently commands a market share of 70.1% as of December 2023 and sees an average of 62,000 new Android mobile apps introduced each month on the Google Play Store, stands as a highly popular platform [2]. Unlike iOS applications, Android apps often do not undergo comprehensive security evaluations [3], underscoring the need to adapt the development process to adhere to rigorous security standards for Android apps.

While diligent requirements analysis and feasibility studies precede development efforts, the final product may still be susceptible to failure due to code vulnerabilities. It is worth emphasising that addressing bugs early in the Software Development Life Cycle (SDLC) is approximately 70 times more cost-effective than addressing them in later stages of the SDLC [4]. Consequently, researchers have devised various automated tools for identifying vulnerabilities in Android apps [5]

* Corresponding author at: School of Computing, Robert Gordon University, Aberdeen, AB10 7QB, United Kingdom.
E-mail addresses: j.senanayake@rgu.ac.uk, janakas@kln.ac.lk (J. Senanayake), h.kalutarage@rgu.ac.uk (H. Kalutarage), a.petrovski@rgu.ac.uk (A. Petrovski), l.piras@mdx.ac.uk (L. Piras), omar.alkadri@udst.edu.qa (M.O. Al-Kadri).

https://doi.org/10.1016/j.jisa.2024.103741

Available online 5 March 2024
2214-2126/© 2024 The Author(s). Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

to prioritise security-focused development and proactively prevent cybersecurity breaches rather than addressing issues later in the app development life cycle.

In previous research, scholars have introduced various supportive tools, frameworks, and plugins aimed at aiding developers in automating the detection process [6]. These tools have utilised both traditional techniques and advanced methods based on Machine Learning (ML) and Deep Learning (DL) to identify vulnerabilities in Android applications. They employ static, dynamic, and hybrid analysis approaches to analyse either the Android Application Package (APK) files or complete Android project source files for vulnerability detection. However, a significant limitation of current solutions is their inability to address the early detection of vulnerabilities in a real-time app development environment. These tools can only assist in identifying vulnerabilities by scanning the code after the development process has concluded.

Leveraging AI-based techniques on a well-annotated dataset of Android source code vulnerabilities can effectively address these limitations. Nonetheless, it is essential to acknowledge the constraints associated with the datasets used to train models for detecting Android vulnerabilities. One feasible approach involves constructing a dataset by labelling the source code after analysing released APKs, but this method has its limitations. The dataset's scope, including the number of distinct vulnerable categories, is restricted, and it may lack sufficient code examples or novel vulnerabilities.

An alternative strategy is to train a model using source code directly obtained from app developers. However, developers might hesitate to share their proprietary code due to privacy concerns [7,8]. To surmount this challenge in the model training process, federated learning can be employed [9]. This approach involves distributing the model training process among multiple entities interconnected within a federated environment. As a result, these entities can independently train the model and make improvements to the final model without revealing their data, which comprises source code samples. Nonetheless, a drawback of the current federated learning method is its limited ability to engage and captivate the participating clients in collaborative training to improve model performances. Hence, a blockchain-based federated learning approach can be implemented to tackle this, wherein model weights are shared within blockchain, and new model updates serve as incentives for those genuinely contributing to enhancing the model's detection capabilities.

Hence, the aim of this paper is to present a novel blockchain-based federated neural network model, incorporating XAI to identify vulnerabilities in Android code with following contributions:

- Evaluating and contrasting the presently available methods and tools for detecting Android code vulnerabilities.
- Introducing Defendroid, a plugin for Android Studio with a neural network-based model as its backend, that excels in accuracy and efficiency for early detection of Android source code vulnerabilities. The initial training of this model utilises the publicly accessible LVDAndro dataset, as described in our previous work [10]. This dataset provides Android source code vulnerabilities, each labelled according to the Mitre CWE [11].
- Incorporating the model with XAI methodologies and generating an API. This API offers explanations for the predictions concerning vulnerable code segments. Android app developers can leverage this information through the plugin to identify potential mitigation strategies.
- Extending the model training within a community-driven, blockchain-powered federated learning setting, aiming to expand and enhance the model's ability to detect vulnerabilities. This approach also helps alleviate the shortage of training data by leveraging a growing network of training nodes, all while addressing concerns related to the privacy of source code.
- Open-sourcing Defendroid and providing access to the public via a GitHub Repository.[1] This repository includes Python scripts and detailed instructions.

The paper follows this structure: In Section 2, the background and relevant prior research are discussed. Section 3 details the vulnerability mitigation model development using blockchain-based federated neural network AI model. Section 4 discusses the applications and capabilities of Defendroid. Finally, Section 5 encompasses the conclusions and outlines future research directions.

2. Background and related work

This section establishes the foundation for the study by elucidating source code vulnerabilities and the ways in which developers can receive support in addressing them. Additionally, it delves into vulnerability scanning methods, AI-driven vulnerability detection, interpretation of prediction outcomes, federated learning, its utility in maintaining privacy during AI model development, and the utilisation of community-driven blockchain-based federated learning. It also reviews pertinent literature on these topics.

2.1. Android developer assistance

According to the research in [12], it has been observed that human errors, often stemming from lapses in focus and concentration, can result in coding issues. Therefore, if there is no extensive testing and validation process in place from the initial stages of the software development lifecycle, such errors can lead to several vulnerabilities in the code [13].

Software developers widely embrace Integrated Development Environments (IDEs) to enhance their productivity in development tasks. These IDEs provide valuable assistance for tasks like code writing, application building, validation, and integration. They come equipped with built-in features and offer the flexibility to incorporate plugins that augment the development process without altering its core functionality. Many of these IDEs also support the installation of third-party plugins developed by external vendors. As discussed in [14], Android app developers also rely on supplementary tools, extensions, and plugins to aid them in their coding endeavours, thus helping to boost their productivity while reducing potential errors.

Google's Android Studio, the official IDE for Android app development which is built upon JetBrain's IntelliJ IDEA, serves as a critical tool in this regard [15]. To assist developers during coding, Table 1 enumerates a selection of beneficial plugins that can be seamlessly integrated into Android Studio.

Though these plugins support various coding activities, none of them focuses on real-time code vulnerability detection. Hence, to introduce such a plugin, developers must possess a strong understanding of source code vulnerabilities.

2.2. Source code vulnerabilities

Source code vulnerabilities encompass inadvertent errors, design flaws, or omissions within the code, all of which may be exploited by malicious actors to compromise the security or functionality of the software. Some of these vulnerabilities are buffer overflows, insecure authentication and authorisation, deserialisation issues, security misconfigurations, and injection vulnerabilities [17]. Developers seeking a comprehensive understanding of these vulnerabilities can refer to the Mitre CWE repository [11].

For mobile application developers also, the Mitre CWE repository serves as a valuable resource to proactively address potential security

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

Table 1
Popular Android Studio plugins for supportive coding [16].

Plugin	Description	Features
ADB Idea	Simplify the android development process by facilitating to perform various essential actions without having to create them from scratch.	Facilitate starting apps, clearing app data, killing apps, restarting apps and uninstalling apps.
Key Promoter X	Provides a mouse-free development experience by guiding users to work with keyboard shortcuts.	Provides a comprehensive list of IDE shortcuts, a catalog of suppressed tips for specific shortcuts, and an easy-to-use feature for creating custom shortcuts for buttons.
Code Glance	Facilitates viewing the complete code block at a glance in a mini-map format.	Highlights code syntax with customised colours and provides an overview of the entire code in a mini-map.
Android Input	Enables easy text input into an Android device or emulator for testing specific features or functionality.	Recalls the last used device and the previously sent text for testing, utilising the Enter key to send a text and the Esc key to refrain from sending.
GitHub Copilot	Delivers AI-assisted auto complete suggestions to enhance the coding experience.	Offers suggestions from GitHub Copilot in two ways: by either initiating code generation based on the developer's input or by describing the desired code functionality through natural language comments.
Tabnine	An AI assisted real-time code completions, chat, and code generation method.	Offers code suggestion, code prediction, code hinting, content assist, unit test generation, and documentation generation.
Space	Streamlines software development, fosters collaboration, and supports team and project management.	Offer assistance for hosting Git repositories, conducting code reviews, automating integration processes, storing and publishing packages, handling issue tracking and documentation, and facilitating team communication via chat.

Table 2
Common vulnerabilities in Android code [11].

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

gaps in their source code. This knowledge proves instrumental in the early detection of vulnerabilities, as demonstrated in [6]. A compilation of common vulnerabilities found in Android code is available in Table 2, categorised based on their likelihood of exploitation.

2.3. Vulnerability scanning techniques

Android applications and their source code must undergo thorough scanning to identify potential issues. The research community has recognised two methods for scanning Android applications, as outlined in [6]: (1) Reverse-engineering APKs to analyse the code, and (2) Real-time analysis of the source code as it is being developed.

When scanning applications, three analysis techniques, static, dynamic or hybrid can be employed. Static analysis identifies code issues without executing the application or the source code, while dynamic analysis necessitates the execution of the application during the scanning process. Hybrid analysis combines aspects of both static and dynamic analysis.

Various tools are available for the analysis of Android applications, such as the Mobile Security Framework (MobSF) [18], a hybrid analysis tool capable of identifying vulnerabilities and malware. The HornDroid tool [19] focuses on analysing information flow within Android apps, and the Quick Android Review Kit (Qark) tool [20] is a static analysis tool designed to detect vulnerabilities in pre-built APKs and complete source code files. However, these supportive tools cannot be seamlessly integrated into the app development environments to assist developers in avoiding vulnerabilities real-time [14].

2.4. AI-powered vulnerability detection

In the quest to identify vulnerabilities in Android source code, various approaches are available, encompassing ML, DL, heuristic-based methods, formal methods, and other non-AI techniques, as discussed in [6]. While traditional methods enjoyed prevalence within the research community during the early stages, there has been a notable surge in the application of AI techniques, as researchers increasingly harness AI to address these challenges, as noted in [5]. The capacity to deliver highly precise outcomes, effortlessly address intricate problems, and offer scalability, coupled with growing popularity, have collectively contributed to the feasibility and potential of developing AI-driven tools to detect Android code vulnerabilities, as suggested in our previous studies [21,22].

In the development of AI-powered tools, the availability of a well-labelled dataset is essential. Numerous datasets have been proposed for this purpose, primarily focused on application vulnerabilities. One

such dataset is Ghera, as detailed in [23], which serves as an open-source benchmark repository encompassing 25 documented vulnerabilities found in Android apps. Ghera additionally provides common characteristics and attributes associated with vulnerability benchmarks and repositories. The National Vulnerability Database (NVD) [24], is another dataset commonly referenced for vulnerabilities. Furthermore, the AndroVul repository, as described in [25], contains a collection of Android security vulnerabilities, including high-risk shell commands, security code anomalies, and details related to dangerous permission-related vulnerabilities.

However, these datasets are insufficient for constructing real-time code vulnerability mitigation methods, as they lack labelling based on actual Android source code. In contrast, the LVDAndro dataset, presented in [10], offers a labelled dataset grounded in the CWE that encompasses Android source code vulnerabilities. The LVDAndro dataset was produced through a combined scanning approach using the MobSF and Qark scanners. The latest LVDAndro APKs combined processed dataset was created by scanning apps from various repositories, including AndroZoo [26], Fossdroid [27], and well-known malware repositories [26]. As the proof of concept for employing ML methods with the LVDAndro dataset to detect Android vulnerabilities has shown promising results [10], it has the potential to serve as a valuable resource for training more advanced AI-based models.

2.5. XAI for interpreting AI prediction results

Conventional AI models typically produce prediction results that are opaque or black boxes. When utilised for vulnerability detection, this lack of transparency poses challenges for app developers, who struggle to comprehend the rationale behind predicted vulnerabilities and to identify potential mitigation strategies. Consequently, developers are compelled to invest additional effort beyond their primary app development tasks.

To overcome this limitation, the application of XAI, as discussed in [28], becomes a viable solution. XAI techniques enhance the transparency of AI models by furnishing human-understandable explanations for their outcomes. These explanations assist users of the model in comprehending the reasoning behind specific decisions or the generation of particular predictions.

Employing XAI can play a pivotal role in pinpointing the root causes of code vulnerabilities. In [29], the authors outline the development of a human-in-the-loop XAI system designed for vulnerability mitigation. This system elucidates model predictions to forensic experts through feature attributions, equipping them with the insights required to make essential corrections. Additionally, the research presented in [30] 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.

In the realm of code vulnerability detection and correction, it is essential to distinguish between factual and counterfactual explanations. A factual explanation addresses the “what” or “why” questions by presenting empirical evidence that substantiates a specific AI model outcome based on the input provided. This explanation also serves to pinpoint the locations of vulnerabilities within the code. In contrast, a counterfactual explanation tackles the “Why-not” or “How-to” questions by crafting a hypothetical scenario that yields a more desirable outcome. This approach aids in demonstrating how to rectify the identified vulnerabilities [31].

Once an AI-powered prediction has been generated, determining the probability of predictions in binary or multi-class classification models can be accomplished through the utilisation of various Python frameworks. Several commonly employed frameworks encompass Shapash, Dalex, Explain Like I’m 5 (ELI5), Local Interpretable Model-agnostic Explanations (LIME), Shapley Additive Explanations (SHAP), and Explainable Boosting Machines (EBM), among others, as detailed in [32]. The choice of framework is contingent on the specific requirements of the prediction task at hand. Hence, it is worth considering the potential applications of these XAI techniques in AI-driven models to detect vulnerabilities in Android code.

2.6. Blockchain federated learning for AI models

Federated learning is grounded in a distributed machine learning approach, which entails training numerous local models on various devices to construct a global model. In a federated environment, clients connecting to the server embark on training their individual local models with their respective data over multiple training rounds. Throughout these rounds, the model weights are transmitted to the federated server, where they are averaged and updated, ultimately culminating in the creation of a global model using the Federated Averaging (FedAvg) algorithm.

FedAvg, as outlined in Algorithm 1, is a widely employed federated learning averaging technique that enables local model training on multiple clients without requiring the sharing of the client’s actual data with the server [33]. The potential for achieving model convergence across diverse client datasets even in non-independent and non-identically distributed settings, is essential in federated learning [34]. Importantly, since federated learning exclusively shares model weights with the federated server, rather than divulging original data, it effectively preserves the privacy of client data [35].

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

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

Nonetheless, traditional federated learning exhibits inherent limitations, particularly in terms of architectural coordination of participants’ activities, the arbitration of their benefits, and the aggregation of models. This is primarily because most existing solutions rely on a centralised approach, necessitating a trustworthy central authority for coordination. Such an approach introduces a multitude of disadvantages, including susceptibility to attacks, a lack of credibility, challenges in calculating rewards, and difficulties in enticing participants for model training. It is for these reasons that blockchain technology has emerged as a potential solution to address the aforementioned issues, as highlighted in [36].

A blockchain, in essence, is a distributed ledger comprised of a sequential series of data blocks, with each block containing a collection of verifiable transactions, as outlined in [37]. Each data block within a blockchain, with the exception of the initial one, incorporates the hash value of the preceding block’s header, creating a linked chain of blocks. Blockchains implement unique consensus mechanisms, such as proof of work (PoW) and proof of stake (PoS), overseen by a Peer-to-Peer (P2P) network of nodes. This setup makes it challenging to generate, yet straightforward to verify each data block. Due to the structural nature of the chain, the complexity involved in block generation, and the decentralised consensus achieved within P2P networks, data within a blockchain is exceedingly resistant to modification. The immutability and traceability of data inherent in blockchains establish them as a

sturdy technical solution for upholding a dependable database in a decentralised and trustworthy manner, as noted in [36].

Consequently, researchers from diverse fields have explored blockchain-based federated learning methods from various angles, as evidenced in [9,38]. However, none of these investigations have thus far delved into the potential application of blockchain with federated learning approaches for AI-based Android code vulnerability detection models.

3. Vulnerability mitigation model development

Current approaches depend on APK files for identifying vulnerabilities in Android code, which presents a challenge in achieving high accuracy during the early stages of app development. Moreover, existing models lack in properly explaining the specific reason for the predicted vulnerability. Therefore, Defendroid seeks to bridge these gaps by employing a blockchain-based federated neural network architecture. This approach facilitates both early and precise detection of Android code vulnerabilities while also delivering explanations for prediction results via XAI. The following section provides an in-depth exploration of the backend model's development journey. It covers essential aspects such as conducting a need analysis, selecting the dataset for model development, constructing the neural network-based model, fine-tuning and pruning model parameters, and expanding the model within a blockchain-based federated learning environment.

3.1. Need analysis

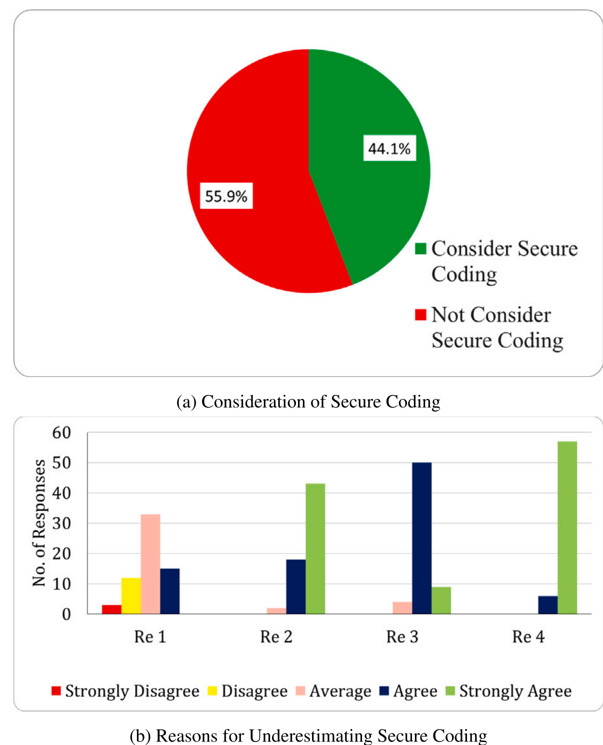
Before embarking on the development of the proposed model, a preliminary step involved conducting a need analysis survey. This survey involved 63 Android app developers employed in app development firms and aimed to determine whether security considerations played a role in their app development practices.

The findings of the survey highlighted that a considerable majority of developers, precisely 55.9%, do not integrate secure coding practices into their app development processes, as illustrated in Fig. 1(a). In the same survey, participants were tasked with rating the factors influencing their limited attention to secure coding, using a 5-Point Likert scale. The outcomes related to these factors are visually represented in Fig. 1(b).

- Re 1. Functionality is more important than security.
- Re 2. Need to allocate additional time to verify the written source code is secured due to rapid development cycles.
- Re 3. Manual verification requires additional resources, including domain experts.
- Re 4. Lack of supportive tools to automate the security checking process.

Following an analysis of the responses, it became evident that a significant number of app developers prioritise both functionality and security, with 33 responses falling into the "Average" rating category. Moreover, a substantial majority of developers, amounting to 68%, firmly advocate allocating extra time for code scrutiny from a security perspective. Additionally, there is a consensus among developers that involving domain experts like security testers and ethical hackers in the development process is essential, particularly when manual security verification is required. This is because developers may lack expertise in identifying source code vulnerabilities and implementing secure coding practices. Furthermore, 91% of the respondents strongly agree that the absence of adequate tool support serves as a major reason for not adequately considering or underestimating security aspects during app development. Consequently, the conclusion was drawn that integrating a highly accurate automated vulnerability detection model into the development pipeline is imperative.

Hence, an open-source solution, Defendroid, has been created to fulfil these requirements. The complete source code and comprehensive instructions are accessible via the Defendroid GitHub repository.[2] An overview of the Defendroid approach is illustrated in Fig. 2.



(a) Consideration of Secure Coding

(b) Reasons for Underestimating Secure Coding

Fig. 1. Survey results.

Initially, Defendroid establishes its backend model through the training of a neural network model using the LVDAndro dataset. This model is then trained within a blockchain federated environment. Subsequently, the best-performing model is seamlessly integrated into a Flask API. This API, in turn, is incorporated into Android Studio as a plugin, enhancing the developer's ability to detect vulnerabilities while leveraging the capabilities of XAI. The detailed steps are elaborated in the following sections.

3.2. Dataset selection

Although there are various vulnerability datasets available, such as those mentioned in [23–25], the majority of them do not specifically address Android source code vulnerabilities or lack comprehensive labelling. In contrast, the LVDAndro dataset, as presented in our previous work, created a labelled dataset based on CWE IDs that specifically focuses on Android code vulnerabilities. This dataset was meticulously crafted through the utilisation of multiple vulnerability scanners and encompasses a substantial dataset of 6,599,597 lines of vulnerable code and 14,689,432 lines of non-vulnerable code, all scanned across 15,021 distinct APKs.

In Table 3, a comprehensive list of the fields encompassed by the LVDAndro dataset can be found. While the processed code, vulnerability status, and CWE IDs are essential for vulnerability detection, other fields can further enrich the dataset with valuable information for prediction. A visual representation of the distribution of CWE-IDs within the LVDAndro Dataset, is depicted in Fig. 3. According to that, it becomes apparent that certain vulnerability categories, like CWE-532 and CWE-312, are frequently observed in source code, whereas examples of code for categories such as CWE-599, CWE-502, and CWE-299 are infrequent.

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

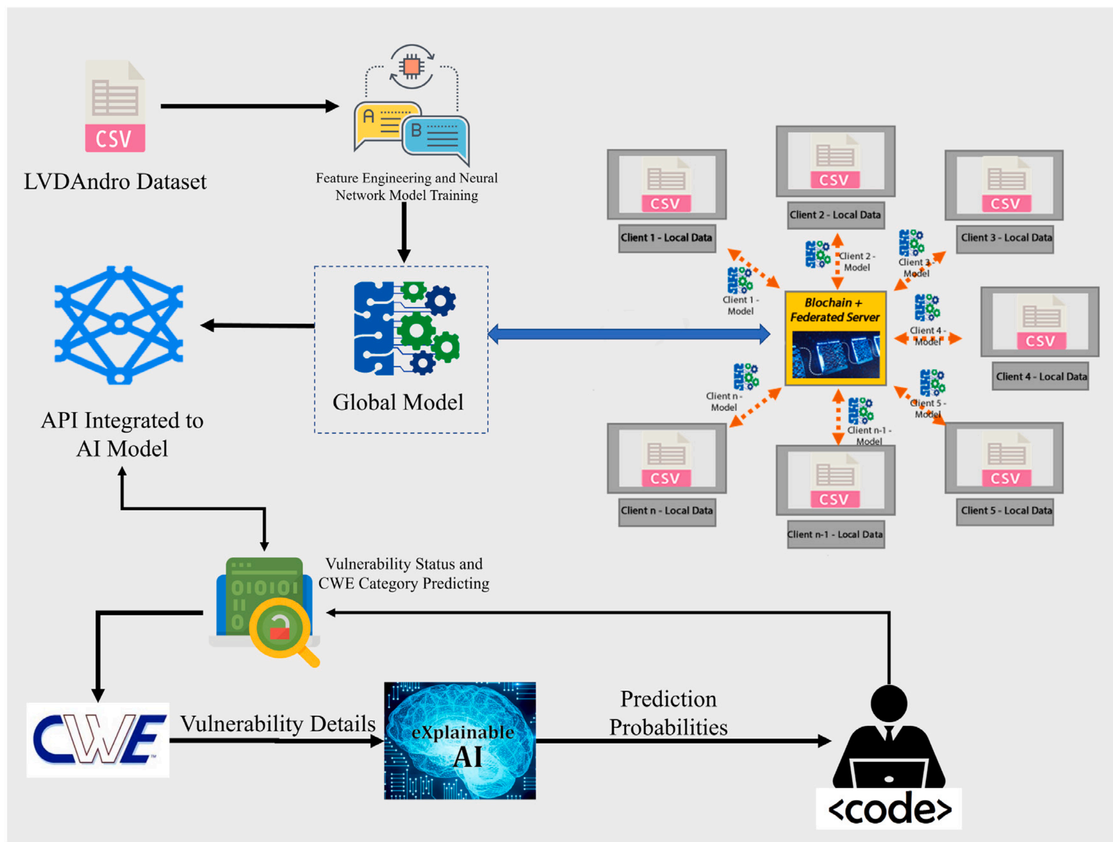


Fig. 2. Overview of the Defendroid model.

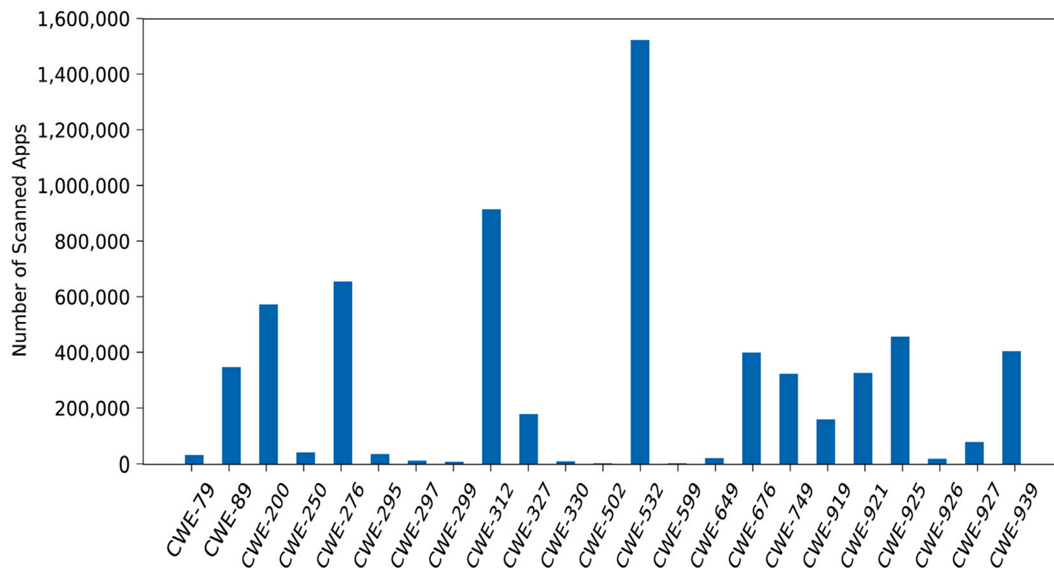


Fig. 3. CWE-ID distribution in LVDAndro dataset.

3.3. Development of neural network-based Vanilla models

LVDAndro provides processed data, and for binary classification analysis, the *processed_code* and *vulnerability_status* fields were employed. To ensure a balanced dataset and mitigate class bias, the initial step involved balancing the dataset with a 1:1 ratio of vulnerable samples to non-vulnerable samples using the NearMiss undersampling technique [39]. Subsequently, the dataset was split, with 75% designated for training and 25% for testing. The construction of the feature vector followed a similar approach as employed in the proof-of-concept of LVDAndro [10], utilising 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 then used to train a neural network model, which included one hidden layer with 20 perceptrons and an output layer with two nodes. The *relu* activation function was employed for the input and hidden layers, while the *sigmoid* activation function was used for the output layer, as it demonstrated favourable performance in experiments. With the aid of a grid search early stopping, with *test_loss*

Table 3
Fields in LVDAndro.

Field name	Description
Index	Auto-generated identifier
Code	Original source code line
Processed_code	Source code line after preprocessing
Vulnerability_status	Vulnerable(1) or Non-vulnerable(0)
Category	Category of the vulnerability
Severity	Severity 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	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

monitoring and parameters set to *min_delta = 0.0001* and *patience = 20* in *auto mode*, was implemented to reduce overfitting. The neural network model's training process utilised the *Adam* optimiser with the default learning rate of *0.001*, and the loss function applied was *binary cross-entropy*.

For multi-class classification, the feature vector was constructed using the *processed_code* and *CWE-ID* fields. Labels were encoded using one-hot encoding. While LVDAndro encompasses code samples for 23 CWE categories, some classes have fewer samples due to their nature. Consequently, only the top 10 classes were retained, and the remaining classes were relabelled as *Other*. The dataset was subsequently 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 used to train a neural network model with an input layer, one hidden layer featuring 20 perceptrons, and an output layer with 11 nodes. The *relu* activation function was employed for the input and hidden layers, while *softmax* was used for the output layer. To reduce overfitting, 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. In the training of the neural network model, the loss function utilised was *categorical cross-entropy*, and the *Adam* optimiser was employed with the default learning rate of *0.001*.

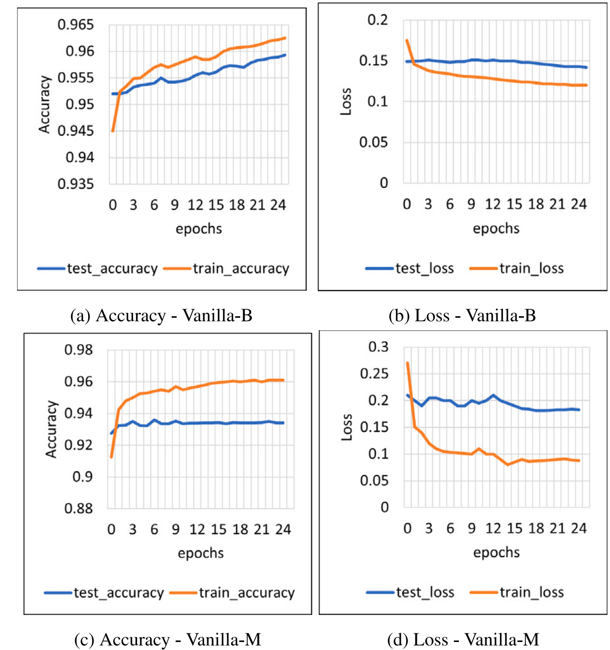
3.3.1. Fine-tuning and pruning of Vanilla model parameters

Numerous experiments were conducted involving the adjustment of model parameters, including variations in the number of hidden layers and the quantity of perceptrons, to identify the optimal configuration. Additionally, a comprehensive grid search and hyper-parameter tuning process were executed to validate the suitability of the aforementioned parameters. After the training phase was completed, an analysis was conducted to assess the F1-Scores and accuracies for both binary and multi-class classification.

Furthermore, pruning techniques were applied to the selected model following parameter tuning. Pruning involves the removal of the least significant weight parameters within a neural network, aiming to enhance throughput while maintaining model accuracy. Magnitude-based pruning serves as a straightforward yet effective method for eliminating weights while preserving the same level of precision. During model training, value zeros are gradually assigned, leading to the gradual removal of inconsequential weights. The model's accuracy depends on the degree of sparsity, necessitating careful selection of the sparsity level to maintain the same level of precision. The implementation of magnitude-based model pruning was facilitated through the utilisation of the TensorFlow model optimisation toolbox [40]. The model was initially trained with all parameters and subsequently pruned to achieve 50% parameter sparsity, commencing from 0% sparsity.

Table 4
Performance comparison of Vanilla models.

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



(a) Accuracy - Vanilla-B (b) Loss - Vanilla-B

(c) Accuracy - Vanilla-M (d) Loss - Vanilla-M

Fig. 4. Accuracy and loss with epochs — Vanilla Models.

3.3.2. Performances of the Vanilla models

Table 4 presents a comparison of F1-Scores, accuracies, and model sizes for both neural network-based binary and multi-class classification tasks. In the context of binary classification, the standard neural network model is denoted as Vanilla-B, while the multi-class classification model is referred to as Vanilla-M. Pruned models, designed for binary and multi-class classifications, are designated as Vanilla-B-P and Vanilla-M-P, respectively.

Based on the data presented in Table 4, it was observed that the unpruned neural network models exhibit slightly better performance when compared to the pruned models. This marginal performance difference can likely be attributed to the number of example codes used and the number of hidden layers employed.

For binary classification models, the variations in training and testing accuracies over epochs are depicted in Fig. 4(a), while Fig. 4(b) visualises the changes in training and testing loss for the same models. In the context of multi-class classification, Fig. 4(c) illustrates the fluctuations in training and testing accuracies over epochs, and Fig. 4(d) presents the training and testing loss profiles.

The best performance results were achieved with 25 epochs for Vanilla-B and 24 epochs for Vanilla-M. For Vanilla-B, the training accuracy reached 96.25%, and the inference accuracy was 95.93%. During this period, the training loss was 0.12, while the testing loss stood at 0.142. As for Vanilla-M, optimal training accuracy of 96.1% and inference accuracy of 93.42% were obtained at the 24-epoch mark, with corresponding training and testing losses of 0.088 and 0.183. The increase in loss observed during training may suggest that the model is becoming overly complex and is potentially fitting noise or outliers in the training data rather than capturing the underlying patterns that are applicable to new data. Given that the unpruned models demonstrate superior performance and the disparities in model

Table 5
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

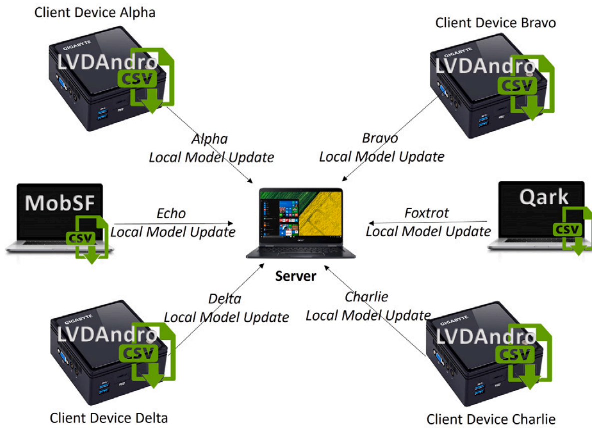


Fig. 5. Federated learning simulated environment.

sizes are insignificant, the decision was made to select these models, namely Vanilla-B and Vanilla-M, for further experiments.

Table 6
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

3.4. Training the model in a federated environment

Federated learning enables the collection of training source code samples from multiple clients while preserving the privacy of their respective code [41]. Hence, a simulated federated learning environment was established in which a server and six clients were utilised, as illustrated in Fig. 5. The server, operating on an Intel Core i5 laptop with 16 GB of RAM and running Windows 11 OS, was responsible for managing the distribution and aggregation of model weights. Four of the clients were running Ubuntu Linux on Gigabyte Brix (GB-BXBT-2807) devices, while the remaining two clients, on Intel Core i5 laptops with 8 GB of RAM, ran Windows 10. These clients were responsible for training models with global weights on their respective local datasets.

Python, TensorFlow, and their associated dependencies were installed on both the server and the clients. The Flower framework [42] was employed, with the server serving as the Flower Server and being connected to the clients. One of the clients, named Alpha, utilised the LVDAndro dataset, while three other clients, Bravo, Charlie, and Delta, employed the LVDAandro dataset generation mechanism to create datasets based on their own data. The client Echo used a dataset generated by scanning APKs using MobSF, and the client Foxtrot used a dataset generated by scanning APKs using Qark. Echo's and Foxtrot's datasets underwent a processing procedure similar to that utilised in LVDAndro [10]. The training datasets for each client contain the records specified in Table 5. In practical scenarios, developers have the flexibility to enhance training by contributing diverse training data obtained through alternative methods, such as manual analysis.

The neural network model parameters in the federated learning model, including the number of hidden layers, neurons, and optimisers, retained the same optimal values as in the Vanilla model. This selected architecture facilitates effective model convergence and entails a federated communication round of 50 along with five epoch iterations, as determined through the fine-tuning process.

After completing 50 rounds of training, the global model was updated on the federated server and is now available for use in the global context. Several federated models were created by varying the clients, to examine the relationship between the number of participating clients and the global model's performance in terms of Accuracy and F1-Score. In the case of binary classification models (Federated-B), these models were denoted as Federated-B-a, Federated-B-ab, Federated-B-abc, Federated-B-abcd, Federated-B-abcde, and Federated-B-abcdef. Here, "a" represents a model trained exclusively with data from client Alpha, "ab" signifies a model trained using data from clients Alpha and Bravo, "abc" involves data from clients Alpha, Bravo, and Charlie, and so forth. For multiclass classification models (Federated-M), a similar naming convention was used. The accuracy and F1-Score of the updated models were compared to the Vanilla models, as detailed in Table 6a and Table 6b.

While comparing the initial Federated binary classification model (Federated-B-a) to the Vanilla binary classification model, it is noteworthy that there was not any discernible improvement in accuracy (improved by 0.03%) and F1-Score (improved by 0.0012). Interestingly, the model's performance slightly improved with 0.16% accuracy and 0.0087 F1-Score improvement when all clients were engaged (Federated-B-abcdef). This lack of notable improvement in model performance when compared to the Vanilla model and other Federated models may be attributed to the fact that Vanilla-B had already been well-trained on a substantial number of samples, while the federated model had a lesser impact.

Conversely, the performance of the multi-class classification models saw significant enhancements in the federated setup. A gradual improvement in performance was evident as more clients' data was integrated into the federated setup for multi-class classification. When all clients were involved in the federated multi-class classification model (Federated-M-abcdef), there was a noteworthy 3.03% increase in accuracy compared to the initial model (Vanilla-M), reaching 96.02%. Furthermore, the F1-Score improved by 0.0519, reaching 0.9624.

In summary, it was evident that the integration of a larger number of clients, particularly when using datasets generated with a similar

approach to LVDAndro, in a federated setup, led to notable performance enhancements in overall model combining the binary and multiclass classifications. Furthermore, building on this proof of concept, clients can practically employ a range of scanning techniques and actively participate in the training process to contribute to model improvements.

3.5. Extending to a blockchain based federated environment

While the federated learning setup can be made available to a broader audience, it might struggle to gain traction due to the absence of incentives for participating clients. Furthermore, there is a need for a mechanism to validate the model's performance to ensure that new client data either improves or at least maintains the current model's effectiveness. To address these challenges, a blockchain-based approach was incorporated alongside the federated model, creating a community-driven model.

However, existing blockchain networks such as Ethereum [43] and Hyperledger Fabric [44] have limitations that hinder their seamless integration with the proposed specialised real-time Android code vulnerability detection model. These limitations include scalability issues, high costs, limited control, lack of rewarding mechanisms, and constraints related to consensus algorithms [45].

Consequently, a dedicated private blockchain was designed to serve the development of the Android code vulnerability detection model. Python and Flask were employed for its creation. In this context, the genesis block was established using a model trained on the LVDAndro dataset. Subsequent blocks are appended to the blockchain network by diligent miners who genuinely contribute to improving the model's performance.

Miniers are still required to be connected to the Federated server for training. However, they are allowed to mine a new block only if they meet the requirements specified by the consensus algorithm as outlined in Algorithm 2. When this is fulfilled, the global model weights are updated, while concurrently maintaining the public ledger with details in SHA-256 encrypted format. Each block is linked to its predecessor by its hash, and it also connects to the subsequent block through hash. The updated model weights and the global model are then shared with the miners who successfully create and integrate a new block into the network.

Algorithm 2: Consensus Algorithm

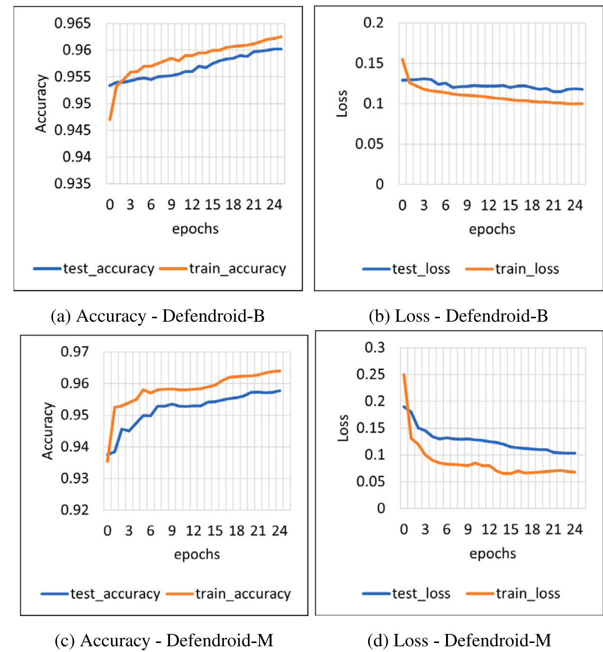
Input: MN: New Model
MC: Current Model
Result: Updates blockchain and global model

1 **if** $MN_{F1\text{-}Score} \geq MC_{F1\text{-}Score}$ and $MN_{Accuracy} \geq MC_{Accuracy}$ **then**
2 Add MN_{Weights} to the blockchain;
3 $MC \leftarrow MN$;

Given that the validation process relies on the employed consensus algorithm, there are no anticipated model performance losses. As a result, the model's accuracy of 96% and F1-Score of 0.96 are consistently expected to improve or at least remain the same when compared to the Federated-B-abcdef and Federated-M-abcdef models. This ultimate model is denoted as "Defendroid" and comprises both binary and multiclass classification models trained within the blockchain-based federated network.

Fig. 6(a) shows the evolution of training and testing accuracies over epochs for Defendroid binary models, while Fig. 6(b) provides a representation of the changes in training and testing loss. Fig. 6(c) displays the variations in training and testing accuracies across epochs, and Fig. 6(d) exhibits the training and testing loss trends for the multi-class model.

The most favourable performance outcomes were observed with 25 epochs for Defendroid-B and 24 epochs for Defendroid-M. For



(a) Accuracy - Defendroid-B (b) Loss - Defendroid-B
(c) Accuracy - Defendroid-M (d) Loss - Defendroid-M

Fig. 6. Accuracy and loss with epochs — Defendroid Models.

Defendroid-B, the training accuracy reached 96.25%, and the inference accuracy was 96.02%. During this phase, the training loss was 0.1, while the testing loss was 0.118. In the case of Vanilla-M, the best training accuracy achieved was 96.4%, and the inference accuracy was 95.78%, both at the 24-epoch mark. The corresponding training and testing losses were 0.068 and 0.103, respectively. These experiments revealed that Defendroid models exhibit a relatively lower degree of overfitting compared to vanilla models, signifying a significant improvement in performance. This may be due to the increased number of code samples involved in model training in the federated learning setup.

4. Applications and capabilities of Defendroid

This section discusses the process of early detecting Android vulnerabilities by employing Defendroid model as an Android Studio plugin.

4.1. Application of Defendroid with XAI

By harnessing the Defendroid API on the backend with the support of XAI, developers gain the capability to promptly detect potential code vulnerabilities while actively coding. This is accomplished by transmitting the code through the API using a seamlessly integrated plugin within their development environment. Consequently, developers can efficiently scrutinise code for vulnerabilities without the need to switch between different applications. This enables them to swiftly pinpoint and address issues as they emerge, facilitating a continuous workflow without interruptions. Such an approach significantly enhances efficiency, conserving both time and valuable resources. For more detailed instructions, please refer to the Defendroid GitHub Repository.[3]

For the binary and multi-class classification models, two pickle files were generated. These files contain the trained model, classifier, and vectoriser components. They were subsequently employed as inputs in the backend of the Flask-based web API for Defendroid, which

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

is developed using Python. The Defendroid web API incorporates a GET request parameter that receives a source code line from a user, subjecting it to a vulnerability check. Upon initialising the web API, the pre-trained binary and multi-class models are loaded from the pickle files.

Upon receiving a user's request via the plugin to the API, the process commences by employing the binary classification vectoriser to transform the code line. Subsequently, the resulting transformed code undergoes assessment by the binary classification model to determine its vulnerability status, classifying it as either vulnerable or non-vulnerable. In cases where the code line is predicted as vulnerable, it then undergoes transformation using the vectoriser associated with the loaded multi-class model. This transformed code is then submitted to the multi-class model for the prediction of the CWE-ID.

Following the predictions of vulnerability status and the CWE-ID, the code line is subjected to processing techniques similar to those used in LVDAndro. This includes procedures like comment replacement and the substitution of user-defined strings. The resultant processed source code is then passed through the Python Lime package, known for its support for XAI. Lime serves the purpose of obtaining insights into the rationales behind the predictions made by both the binary and multi-class models. This information is conveyed in the form of prediction probabilities.

The Lime package delivers insights into the contributions of individual words within the processed source code line, elucidating their significance in both the vulnerability prediction and the prediction of the vulnerable category. Subsequently, the prediction results are transmitted from the API in the form of a JSON response and are subsequently relayed to the plugin.

4.1.1. Plugin integration to Android studio

The most recent version of the plugin is distributed in the form of a JAR file, which can be obtained from GitHub.[4] To seamlessly integrate this JAR file into the latest version of Android Studio, the developers can simply follow the standard procedure for installing a third-party plugin in the Android Studio IDE. Additionally, for compatibility with older versions of Android Studio, adjustments can be made by modifying the version specification accordingly[5] in the plugin.xml file. Once the plugin is successfully installed, it provides vulnerability-related suggestions as balloon notifications. The plugin offers two vulnerability detection options which can be used while code is being written.

- **Quick Check**: This option involves scanning the entire source code file to identify the presence of vulnerable source code.
  To initiate the quick check option, the developer can navigate to *Tools → Check Source Vulnerability*, or simply use the shortcut *CTRL+ALT+E* within Android Studio. This option provides a swift means to identify vulnerable code lines and their respective CWE IDs in a single scan, offering a rapid vulnerability assessment.
- **Detailed Check**: In this option, the plugin determines whether any vulnerabilities are linked to a specific code line.
  For the detailed check option, the developer can access it through *Tools → Check Code Vulnerability*, or use the shortcut *CTRL+ALT+A* when the cursor is placed on a specific code line.

These options are visually represented in Fig. 7.

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

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

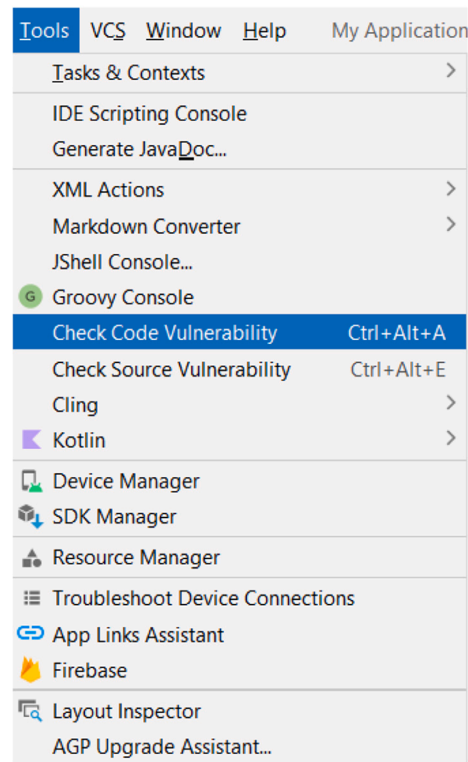


Fig. 7. Defendroid integration with Android studio in Tools menu.

4.1.2. Plugin usage

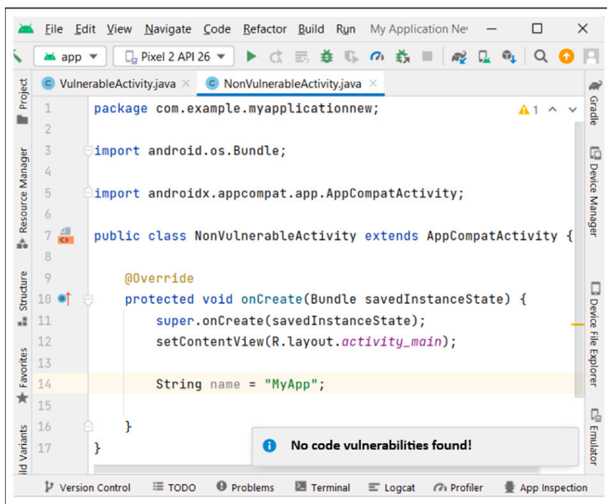
The balloon notification generated by the Defendroid plugin conveys the outcomes derived from the API. After conducting a quick scan, developers are presented with a balloon notification indicating the status of vulnerable code within the source file. If no vulnerable code is detected, a notification resembling Fig. 8(a) is displayed. Conversely, if vulnerable code segments are found, the notification, as shown in Fig. 8(b), highlights the presence of such code lines along with their corresponding CWE IDs. For a closer look, Fig. 8(b) is further magnified in Fig. 8(c).

During a detailed check, the developer will receive a notification that provides the status of the source code's vulnerability. If the code is found to be non-vulnerable, the developer will receive a notification resembling Fig. 9(a). The focus here is on the statement *String name = "MyApp"*.

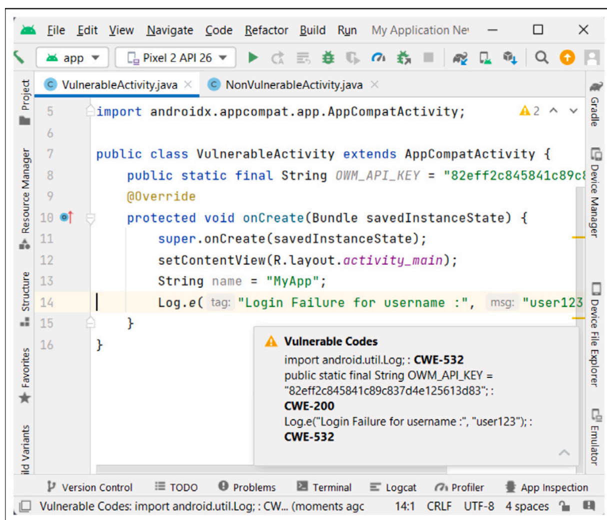
In the event that the code currently under the cursor is identified as vulnerable, a balloon notification will be presented as in Fig. 9(b). This notification includes a detailed description of the vulnerability, accompanied by guidance on how to mitigate it. The notification also features information regarding the probability of vulnerability status prediction (binary classification), the associated CWE ID, and the prediction probability within the CWE category (multi-class classification prediction). Additionally, it provides insights into the contributions of individual words to the probability in both binary and multi-class classification approaches.

The type of notification, whether informational or a warning, is contingent upon the severity of the vulnerability. Furthermore, to offer more comprehensive information about the vulnerability, Defendroid suggests methods for addressing it, often by referring to the CWE repository [11].

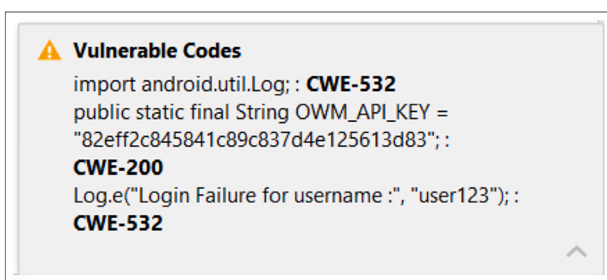
For a practical example of a detailed check on a vulnerable code line, please refer to Fig. 9(b). To examine the specific balloon notification in more detail, see Fig. 9(c). In this example, the cursor is focused on the statement *Log.e("Login Failure for username:", "user123");*. This



(a) Notification of Non-vulnerable Source File

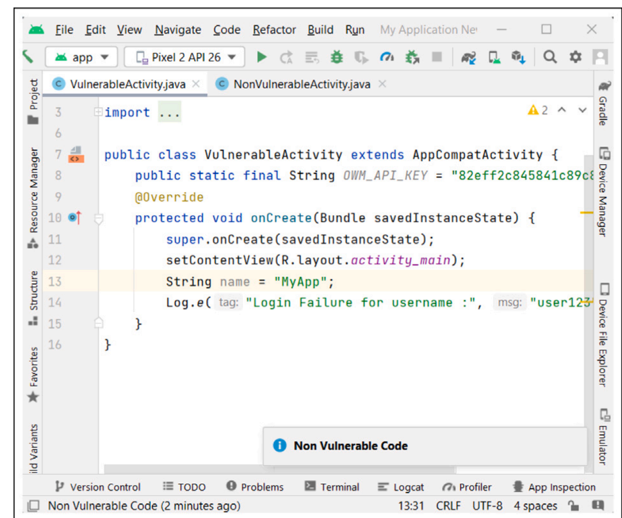


(b) Notification of Vulnerable Source File

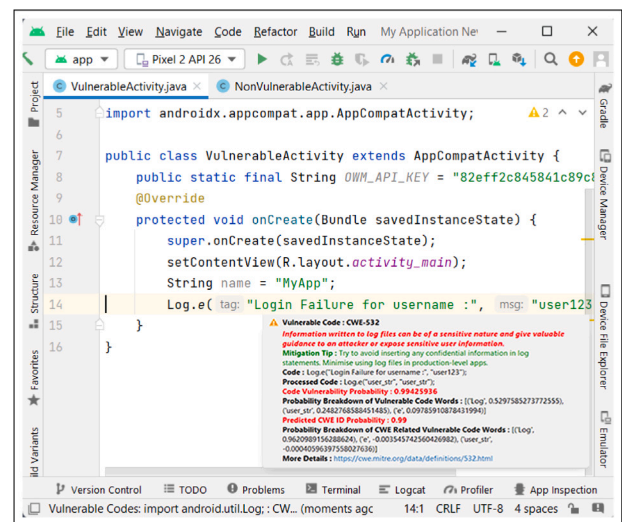


(c) Balloon Notification

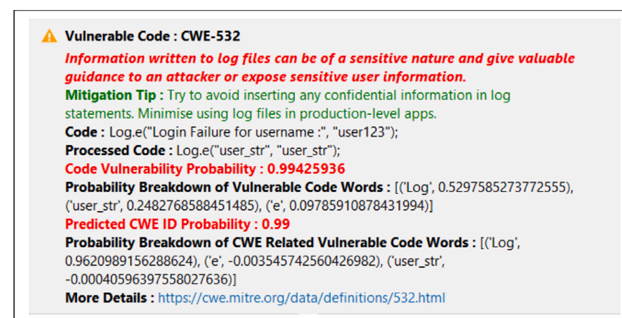
Fig. 8. Defendroid quick check notifications.



(a) Notification for a Non-Vulnerable Code Line



(b) Notification for a Vulnerable Code Line



(c) Balloon Notification

Fig. 9. Defendroid detailed check notifications.

particular code is linked to CWE-532, a classification that the model accurately predicted with a high probability of 0.99. Notably, within this prediction, the model identified “Log” as the most influential element, attributing it with a substantial probability of 0.53. In the context of multi-class classification, the model exhibited a prediction probability of 0.99 for CWE-532, and it attributed a significant contribution of 0.96 to the term “Log”. This emphasises the importance of developers exercising caution when incorporating log statements into production-level applications. Such statements can potentially introduce vulnerabilities that attackers may exploit by inspecting log files. As a preventive measure, developers can implement encryption processes to generate log files in an encrypted format, rather than plain text, enhancing security and mitigating potential risks.

The Android Studio Event Log is synchronised with these executions, as depicted in Fig. 10. This feature aids developers in monitoring the progression of source code in the context of vulnerability mitigation.

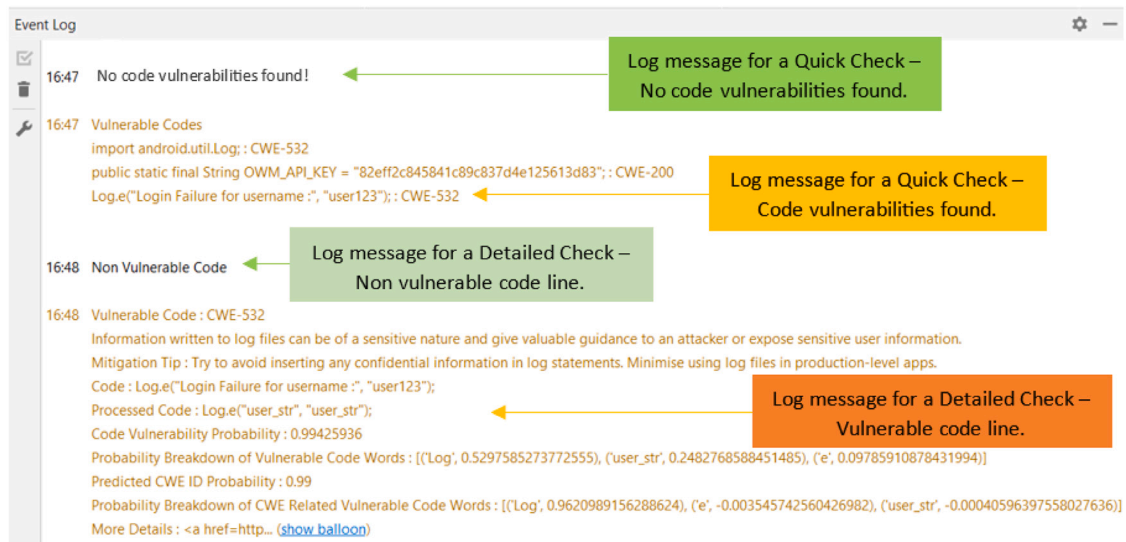


Fig. 10. Defendroid notifications in Event Log.

Table 7
Performance comparison of MobSF, Qark and Defendroid models.

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**

Developers are not obliged to follow a strict order when utilising the quick or detailed check options; they can choose the sequence that best suits their preferences. Empowered by these recommendations and the accompanying prediction probabilities, developers are equipped to enhance the security of their apps by addressing source code vulnerabilities. The ability to revisit the vulnerability assessment provides developers with the advantage of monitoring the fluctuations in prediction probabilities when specific code lines are altered. This functionality proves beneficial in scenarios where complete mitigation may not be feasible, such as situations where retaining log file records for debugging purposes is essential, even in production-level apps.

4.2. Capabilities of Defendroid

Defendroid possesses the capability to identify 10 CWE categories (11 when including the "other" category), all of which are characterised by either a high or medium likelihood of exploitation, as elaborated in [21]. These CWE-IDs encompass 89, 200, 276, 312, 532, 676, 749, 921, 925, and 939.

Therefore, to assess the accuracy of the Defendroid model, a comparative analysis was conducted against the MobSF and Qark scanners, both of which contributed to the construction of the initial LVDAndro dataset. The evaluation aimed to gauge the accuracy of identifying vulnerable code within new data, involving a total of 2216 source code lines. This compilation featured randomly selected 604 lines of vulnerable code examples derived from the CWE repository, as well as 1612 lines of non-vulnerable code extracted from real applications. These code lines were seamlessly incorporated into an Android app project, subsequently subjected to scanning using both MobSF and Qark Scanners.

These same code lines were then presented to the Defendroid model through its Android Studio plugin. The comparative evaluation encompassed metrics such as accuracy, precision, recall, and F1-Score, the results of which are comprehensively summarised in Table 7.

Upon conducting the comparative analysis, it became evident that Defendroid outperformed MobSF and Qark in its ability to predict vulnerabilities. Defendroid achieved a high accuracy rate of 96%, coupled with a precision score of 0.95, a recall rate of 0.99, and an F1-Score of 0.96. Notably, Defendroid excelled in reducing the false negative rate, thereby mitigating potential security risks associated with its predictions.

The Defendroid plugin can predict a single line of code in under 300 ms. Its performance was assessed by measuring the time it took for prediction after integrating it with Android Studio on a Windows OS system with a Core i5 processor and 16 GB RAM. However, initialising the API in this environment took between 3 to 6 s and consumed about 100MB of memory before the plugin could execute. This initialisation is a one-time process, meaning developers will not need to spend extra time or effort to obtain real-time prediction results afterward.

Defendroid's functionalities were also contrasted with various well-known tools and techniques employed for vulnerability detection, as illustrated in Table 8.

Furthermore, it is anticipated that the model's performance will witness continual enhancements and reach optimal levels as this blockchain-based federated learning environment becomes accessible to a broad spectrum of clients and miners, ranging from individual app developers to app development companies.

4.3. Developer feedback

The Android app developers who took part in the preliminary needs assessment survey (Section 3.1) were provided with the plugin for integration into Android Studio, which they employed in the course of app development. Subsequently, a survey was administered to collect feedback concerning the plugin's performance. Developers were requested to express their satisfaction levels using a 5-Point Likert scale. With the help of domain experts, the developers were asked to manually analyse and verify the prediction results generated by the plugin. The results of this survey, gathered from 63 developers, are presented graphically in Fig. 11.

The survey findings unveiled that a significant majority, encompassing 87% of app developers, held a high level of satisfaction regarding the accuracy and efficiency of Defendroid's predictions. Moreover, an impressive 89% of developers expressed high satisfaction with Defendroid's overall usefulness and the quality of its mitigation recommendations.

Nonetheless, the survey underscored opportunities for enhancing the plugin's usability and integration features, as only approximately

Table 8
Highlights of Defendroid and other popular vulnerability detection tools.

(a) Detection capabilities, computational requirements, integration capabilities, user friendliness and other features

Tool/Method	Detection capabilities and computational requirements	Integration capabilities and user friendliness	Other features
Snyk [46]	* Claiming 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 [47]	* Claiming 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 [48]	* 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 [49]	* 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 retraining 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

(b) Summary

Feature	MobSF [18]	Qark [20]	Snyk [46]	Immuni Web [47]	Drozer [48]	Astra [49]	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	–	–	–	–	–	–	–	✓

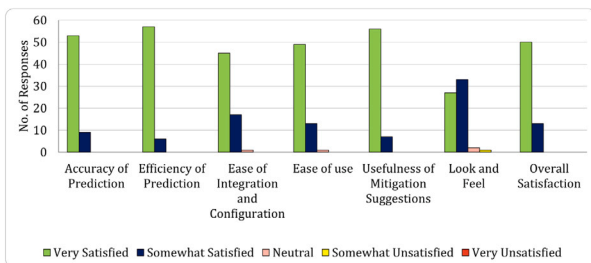


Fig. 11. Survey results — Defendroid satisfaction.

22% of developers reported a high level of satisfaction in these areas. Additionally, there is room for improvement in terms of the plugin's visual aesthetics, given that 57% of developers were not highly satisfied with its look and feel. This feedback is particularly valuable as it can serve as a basis for enhancing the plugin's appeal. One approach could involve integrating mitigation suggestions akin to IDEs' syntax error indication features, which would involve highlighting issues and providing recommendations in a more intuitive manner, rather than through balloon notifications.

Notwithstanding the identified areas for improvement, the overall satisfaction rate remains remarkably high, with 79% of developers expressing a high degree of satisfaction and an additional 21% reporting moderate satisfaction. With further development and refinement, the plugin exhibits the potential to gain wider adoption within the developer community for the purpose of mitigating Android source code vulnerabilities.

5. Conclusion and future work

The adoption of secure coding practices is crucial when developing Android apps. This study introduces Defendroid, an innovative approach to detect vulnerabilities during the source code writing process. Trained on the LVDAndro dataset, the neural network model integrates with a blockchain-based federated learning environment, collaborating with local models to enhance detection capabilities and overall accuracy. The collaborative learning process includes model performance validation through the block mining stage governed by the consensus algorithm. Additionally, XAI is incorporated to offer comprehensive insights into the rationale behind vulnerability predictions, providing valuable support for developers.

Defendroid boasts impressive results, achieving a 96% accuracy rate and an F1-Score of 0.96 for both binary and multiclass classifications. Developers can seamlessly employ the prototype plugin for Android Studio to proactively detect and fix vulnerabilities using Defendroid. The Defendroid model is accessible via its GitHub repository. A plugin prototype has been created and employed, to demonstrate the capabilities of this approach. However, exploring the incorporation of the Defendroid model and its API as a full product, as a user-friendly Android Studio plugin, could be a valuable avenue for future development. By incorporating principles of XAI and harnessing the capabilities of federated learning, Defendroid advances the field of Android code vulnerability detection while preserving the privacy of source code while training it. As the environment is made available to a broader audience and transitions to a fully community-driven, decentralised model, the model's accuracy is expected to see further improvements in the future. Securing the transmission and sharing of weights within the blockchain can enhance the overall security of the model and this aspect can be investigated in more detail while introducing an incentive mechanism to attract more clients in future research.

CRediT authorship contribution statement

Janaka Senanayake: Conceptualization, Data curation, Methodology, Software, Validation, Writing – original draft. **Harsha Kalutarage:** Conceptualization, Supervision, Writing – review & editing. **Andrei Petrovski:** Conceptualization, Supervision, Writing – review & editing. **Luca Piras:** Conceptualization, Supervision, Writing – review & editing. **Mhd Omar Al-Kadri:** Conceptualization, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Source code and data are available at Defendroid GitHub repository.

Acknowledgements

We thank Robert Gordon University, United Kingdom, the AHEAD grant and University of Kelaniya, Sri Lanka for their support.

References

[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. http://dx.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 november 2023. 2023, https://www.statista.com/statistics/1020956/android-app-releases-worldwide/.

[3] Garg S, Baliyan N. Comparative analysis of android and iOS from security viewpoint. Comp Sci Rev 2021;40:100372. http://dx.doi.org/10.1016/j.cosrev.2021.100372.

[4] Krasner H. The cost of poor software quality in the US: A 2020 report. In: Proc. consortium inf. softw. qualityTM. 2021, URL 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 Comput Surv 2017;50(4). http://dx.doi.org/10.1145/3092566.

[6] Senanayake J, Kalutarage H, Al-Kadri MO, Petrovski A, Piras L. Android source code vulnerability detection: A systematic literature review. ACM Comput Surv 2023;55(9). http://dx.doi.org/10.1145/3556974.

[7] 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. http://dx.doi.org/10.1007/978-3-030-58986-8_13.

[8] 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. Inf Comput Secur 2020;28(4):531–53. http://dx.doi.org/10.1108/ICS-01-2020-0002.

[9] Li L, Fan Y, Tse M, Lin K-Y. A review of applications in federated learning. Comput Ind Eng 2020;149:106854. http://dx.doi.org/10.1016/j.cie.2020.106854.

[10] 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. SciTePress, INSTICC; 2023, p. 659–66. http://dx.doi.org/10.5220/0012060400003555.

[11] MITRE Corporation. Common weakness enumeration (CWE). 2023, URL https://cwe.mitre.org/.

[12] Nagaria B, Hall T. How software developers mitigate their errors when developing code. IEEE Trans Softw Eng 2022;48(6):1853–67. http://dx.doi.org/10.1109/TSE.2020.3040554.

[13] 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. http://dx.doi.org/10.1007/978-3-031-32636-3_16.

[14] Tang J, Li R, Wang K, Gu X, Xu Z. A novel hybrid method to analyze security vulnerabilities in android applications. Tsinghua Sci Technol 2020;25(5):589–603. http://dx.doi.org/10.26599/TST.2019.9010067.

[15] Google. Android studio. 2023, URL https://developer.android.com/studio.

[16] JetBrains. Android studio plugins and themes. 2023, URL https://plugins.jetbrains.com/androidstudio.

[17] 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. Montreal, QC, Canada: IEEE; 2019, p. 383–7. http://dx.doi.org/10.1109/MSR.2019.00064.

[18] Abraham A, Magaofei, Dobrushin M, Nadal V. Mobile security framework (mobsf). 2023, URL https://github.com/MobSF/Mobile-Security-Framework-MobSF.

[19] 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. http://dx.doi.org/10.1109/EuroSP.2016.16.

[20] LinkedIn. Quick android review kit (QARK). 2015, URL https://github.com/linkedin/qark/.

[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. http://dx.doi.org/10.1007/978-3-031-37586-6_20.

[22] Senanayake J, Kalutarage H, Al-Kadri MO, Petrovski A, Piras L. Fedrevan: Real-time detection of vulnerable android source code through federated neural network with XAI. In: Computer security. ESORICs 2023 international workshop. Cham: Springer Nature Switzerland; 2023, p. 1–16. http://dx.doi.org/10.1007/978-3-031-54129-2_25.

[23] Mitra J, Ranganath V-P. 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. http://dx.doi.org/10.1145/3127005.3127010.

[24] NIST. National vulnerability database. 2021, URL https://nvd.nist.gov/vuln.

[25] 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, URL https://dl.acm.org/doi/abs/10.5555/3370272.3370279.

[26] 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. New York, NY, USA: ACM; 2016, p. 468–71. http://dx.doi.org/10.1145/2901739.2903508.

[27] Simonin D. Fossdroid. 2022, URL https://fossdroid.com/.

[28] 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. 2022, http://dx.doi.org/10.48550/ARXIV.2206.03585.

[29] 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. 2021, p. 1210–2. http://dx.doi.org/10.1109/ASE51524.2021.9678840.

[30] Wijekoon A, Wiratunga N. A user-centred evaluation of DisCERN: Discovering counterfactuals for code vulnerability detection and correction. Knowl-Based Syst 2023;278:110830. http://dx.doi.org/10.1016/j.knosys.2023.110830, URL https://www.sciencedirect.com/science/article/pii/S0950705123005804.

[31] Miller T. Explanation in artificial intelligence: Insights from the social sciences. Artificial Intelligence 2019;267:1–38. http://dx.doi.org/10.1016/j.artint.2018.07.007, URL https://www.sciencedirect.com/science/article/pii/S0004370218305988.

[32] Bhatnagar P. Explainable AI (XAI) — A guide to 7 packages in python to explain your models. 2021, https://towardsdatascience.com/explainable-ai-xai-a-guide-to-7-packages_in-python-to-explain-your-models-932967f0634b.

[33] Li T, Sahu AK, Talwalkar A, Smith V. Federated learning: Challenges, methods, and future directions. IEEE Signal Process Mag 2020;37(3):50–60. http://dx.doi.org/10.1109/MSP.2020.2975749.

[34] Li X, Huang K, Yang W, Wang S, Zhang Z. On the convergence of FedAvg on non-IID data. 2020, arXiv:1907.02189.

[35] 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. http://dx.doi.org/10.1007/978-3-031-21311-3_6.

[36] Zhu J, Cao J, Saxena D, Jiang S, Ferradi H. Blockchain-empowered federated learning: Challenges, solutions, and future directions. ACM Comput Surv 2023;55(11). http://dx.doi.org/10.1145/3570953.

[37] 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. 2018, p. 49–56. http://dx.doi.org/10.1109/SMARTCOMP.2018.00073.

[38] Qu Y, Uddin MP, Gan C, Xiang Y, Gao L, Yearwood J. Blockchain-enabled federated learning: A survey. ACM Comput Surv 2022;55(4). http://dx.doi.org/10.1145/3524104.

[39] Soltanzadeh P, Feizi-Derakhshi MR, Hashemzadeh M. Addressing the class-imbalance and class-overlap problems by a metaheuristic-based under-sampling approach. Pattern Recognit 2023;143:109721. http://dx.doi.org/10.1016/j.patcog.2023.109721, URL https://www.sciencedirect.com/science/article/pii/S0031320323004193.

[40] Google. TensorFlow model optimization. 2023, URL https://www.tensorflow.org/model_optimization.

[41] 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. http://dx.doi.org/10.1007/978-3-030-27813-7_6.

[42] Beutel DJ, Topal T, Mathur A, Qiu X, Fernandez-Marques J, Gao Y, et al. Flower: A friendly federated learning research framework. 2022, arXiv:2007.14390.

[43] Vujicic D, Jagodic D, Randic S. Blockchain technology, bitcoin, and ethereum: A brief overview. In: 2018 17th international symposium INFOTEh-JAHORInA. 2018, p. 1–6. http://dx.doi.org/10.1109/INFOTEH.2018.8345547.

[44] 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. New York, NY, USA: Association for Computing Machinery; 2018, p. 1–15. http://dx.doi.org/10.1145/3190508.3190538.

[45] 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. http://dx.doi.org/10.1016/B978-0-12-819816-2.00013-7, URL https://www.sciencedirect.com/science/article/pii/B9780128198162000137.

[46] Snyk. Snyk developer security. 2023, URL https://snyk.io/.

[47] immuniweb. Mobile application penetration testing | ImmuniWeb mobile suite. 2023, (September 23). 2023, URL https://www.immuniweb.com/products/mobile/.

[48] WithSecure. The leading security testing framework for android. 2023, URL https://labs.withsecure.com/tools/drozer.

[49] Sharma S, Krishna A. Hacker style pentest by astra security. 2023, URL https://www.getastra.com/pentest.