Android Code Vulnerabilities Early Detection using AI-powered *ACVED* Plugin

Janaka Senanayake[1,2][0000−0003−2278−8671],
Harsha Kalutarage[1][0000−0001−6430−9558],
Mhd Omar Al-Kadri[3][0000−0002−1146−1860],
Andrei Petrovski[1][0000−0002−0987−2791], and
Luca Piras[4][0000−0002−7530−4119]

[1] School of Computing, Robert Gordon University, Aberdeen AB10 7QB, UK
{j.senanayake,h.kalutarage,a.petrovski}@rgu.ac.uk
[2] Faculty of Science, University of Kelaniya, Kelaniya, Sri Lanka
janakas@kln.ac.lk
[3] School of Computing and Digital Technology, Birmingham City University, Birmingham B5 5JU, UK
omar.alkadri@bcu.ac.uk
[4] Department of Computer Science, Middlesex University, London NW4 4BT, UK
l.piras@mdx.ac.uk

Abstract. During Android application development, ensuring adequate security is a crucial and intricate aspect. However, many applications are released without adequate security measures due to the lack of vulnerability identification and code verification at the initial development stages. To address this issue, machine learning models can be employed to automate the process of detecting vulnerabilities in the code. However, such models are inadequate for real-time Android code vulnerability mitigation. In this research, an open-source AI-powered plugin named **A**ndroid **C**ode **V**ulnerabilities **E**arly **D**etection (ACVED) was developed using the LVDAndro dataset. Utilising Android source code vulnerabilities, the dataset is categorised based on Common Weakness Enumeration (CWE). The ACVED plugin, featuring an ensemble learning model, is implemented in the backend to accurately and efficiently detect both source code vulnerabilities and their respective CWE categories, with a 95% accuracy rate. The model also leverages explainable AI techniques to provide source code vulnerability prediction probabilities for each word. When integrated with Android Studio, the ACVED plugin can provide developers with the vulnerability status of their current source code line in real-time, assisting them in mitigating vulnerabilities. The plugin, model, and scripts can be found on GitHub, and it receives regular updates with new training data from the LVDAndro dataset, enabling the detection of novel vulnerabilities recently added to CWE.

Keywords: Android application security · code vulnerability · labelled dataset · artificial intelligence · plugin

1 Introduction

As of March 2023, the Google Play Store sees an average of 87,000 new Android mobile apps released each month, and Android dominates the market with a 70.93% share [23, 24]. However, due to the lack of adherence to secure coding practices and standards, some of these apps have source code vulnerabilities that are attractive to hackers [26]. Therfore, the security of Android apps may always not be guaranteed. Hence, it is important to mitigate vulnerabilities promptly. It is worth noting that delaying bug fixing until later stages in the Software Development Life Cycle (SDLC) is 30 times more expensive than fixing them early on [9].

Without proper mechanisms in place, developers may not consider potential vulnerabilities of the source code. However, developers should pay attention to this as identifying source code weaknesses at an early stage can make the software less vulnerable. Therefore, it is essential to support Android app developers to continuously prioritise and apply security best practices. Developers find supportive tools, frameworks, and plugins beneficial in automating the coding process. However, there is a shortage of automated supportive tools that follow security best practices and address code vulnerabilities during application development [19].

Despite a few available tools that use conventional methods, Machine Learning (ML) methods and Deep Learning (DL) methods with static, dynamic, and hybrid analysis to identify Android app vulnerabilities [7, 21], several limitations such as fewer detection capabilities and low performances, exist. Additionally, these tools cannot perform real-time detection during coding, and they can only identify vulnerabilities by analysing either Android Application Package (APK) files or the entire source code of an Android project.

To address such limitations, this paper makes the following contributions:

- An Artificial Intelligence (AI) based model that employs ensemble learning techniques to detect Android source code vulnerabilities with high accuracy. The LVDAndro dataset [20] was used to train this model.
- A plugin named ACVED, which employs the trained model and Explainable AI (XAI) techniques in its backend. The plugin can be integrated with Android Studio and can assist developers in identifying potential vulnerabilities in the source code and recommend appropriate mitigating approaches based on the reasoning behind the predictions.
- The plugin and model are publicly available as a GitHub Repository[1] along with source code and essential instructions for improvement to address the latest potential vulnerabilities.

The following is the organisation of the paper: In Section 2, background and related work are discussed, while the development process of the vulnerability detection model is explained in Section 3. The application of the ACVED Plugin is described in Section 4. The conclusion and future works are discussed in Section 5.

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

2 Background and Related Work

In this section, the foundation for the study is established by examining several topics and exploring related studies on source code vulnerabilities, techniques, tools and frameworks for vulnerability scanning and analysis, datasets used for machine learning (ML)-based vulnerability detection models, and the use of XAI to understand predictions. Additionally, how assistive tools can support Android developers in their work is also discussed.

2.1 Source Code Vulnerabilities

Reducing vulnerabilities in the source code is crucial for promoting secure software development practices, as highlighted in [16, 27]. However, without proper mechanisms, developers may overlook potential vulnerabilities. Therefore, organisations and the community have identified various vulnerabilities documented in repositories such as Common Weakness Enumeration (CWE) [3] and Common Vulnerabilities and Exposures (CVE) [4], which are widely used. These repositories contain software and hardware-related vulnerabilities that can be identified across many platforms, making them a valuable resource for mobile application developers to mitigate security loopholes by identifying patterns in their source code. If automated tool support based on CWE and CVE details were available, the Android app development process could be completed efficiently by minimising vulnerable source code.

2.2 Application Analysis and Vulnerability Scanning

To identify source code vulnerabilities in Android apps, the first step involves analysing and scanning the completed app or the source code. Two approaches are available for scanning Android apps for source code vulnerabilities: 1) scanning the code by reverse-engineering the developed APKs, and 2) scanning the source code simultaneously as the code is being written [17]. Since the first approach requires a pre-built application, it cannot be applied in the early stages of the software development life cycle (SDLC) [1]. Although the second approach is more valuable to developers as it focuses on early detection, it is not widely practised due to the limited availability of tools and frameworks.

To analyse applications, static, dynamic, and hybrid analysis techniques can be employed. Static analysis methods can identify code issues without executing the application or the source code. Two types of static analysis techniques are available: Manifest analysis and code analysis. Manifest analysis can extract features for analysis by identifying package names, permissions, activities, services, intents, and providers. In contrast, code analysis can provide more insights into the source code by analysing features such as API calls, information flow, native code, taint tracking, clear-text analysis, and opcodes [8]. Conversely, dynamic analysis requires a runtime environment to execute the application for scanning. Five feature extraction methods for dynamic analysis have been identified: code

instrumentation, system resources analysis, system call analysis, network traffic analysis, and user interaction analysis [6]. The hybrid analysis combines the features of both static and dynamic analysis. By performing a hybrid analysis, a mix of static and dynamic features can be extracted [6].

2.3 Tools and Frameworks for Vulnerability Scanning

There are several tools available for vulnerability scanning of Android applications, as discussed in [19]. Some widely used free tools include Qark, MobSF, AndroBugs, DevKnox, and JAADAS, and these can be utilised by app developers to detect source code vulnerabilities using application scanning techniques. Additionally, tools such as Guardsquare, AppSweep, DeepSource, Copilot and SonarQube are available, offering similar services, but they are not free to use. Previous research has suggested the application of ML-based and non-ML-based methods for this purpose, with a recent trend of using ML-based methods [7]. The accuracy and performance of ML models can be enhanced through dataset improvements and parameter tuning, as seen in experiments conducted in various studies, including [5, 10, 18, 28]. These studies used ML algorithms such as Decision Tree (DT), Naive Bayes (NB), AdaBoost (AB), ID3, C4.5, J48, K-Star, Random Forest (RF), Gradient Boosting (GB), Extreme Gradient Boosting (XGB), Logistic Regression (LR), Support Vector Machine (SVM), and Multi-Layer Perception (MLP). However, none of these methods proposed a real-time Android code vulnerability detection approach.

2.4 Datasets for ML-based Vulnerability Detection

ML-based models can be developed by training ML algorithms on properly labelled datasets. Several datasets have been proposed primarily related to application vulnerabilities. For instance, Ghera [12] is an open-source benchmark repository that captures 25 known vulnerabilities in Android apps. It also outlines some common characteristics of vulnerability benchmarks and repositories. The National Vulnerability Database (NVD) [15] is another dataset used to reference vulnerabilities. The AndroVul repository [14] contains Android security vulnerabilities, including high-risk shell commands, security code smells, and dangerous permission-related vulnerability details. LVDAndro [20], an Android code vulnerability dataset, contains vulnerable and non-vulnerable source code labelled with CWE categories. The proof-of-concept demonstrates that the LVDAndro is applicable in training ML models to detect Android code vulnerabilities.

2.5 Understanding ML-based Vulnerability Predictions with XAI

ML models often function as a black box, providing only the output predictions and not the reasoning behind them. This lack of transparency can make it difficult for developers to identify the underlying causes of vulnerabilities and devise

appropriate mitigation strategies. To better understand the reasons behind these predictions, additional effort outside of app development is often necessary [22]. XAI can help with this by generating algorithms that are both accurate and explainable [11]. Various Python-based frameworks, such as Shapash, Dalex, ELI5, Lime, SHAP, and EBM, can provide the probability of predictions in binary or multi-class classification models [2]. These frameworks can be selected based on specific requirements, such as Lime's applicability for text or image classification-related predictions.

2.6 Tools for Assisting Android Developers

The study in [13] suggests that code issues can arise due to human errors like a lack of focus and concentration. To address this, software developers use various strategies like self-concentration, process checklists, and integrated tools during development. To improve efficiency, developers often rely on Integrated Development Environments (IDEs) that assist with code writing, application building, validation, and integration. IDEs have built-in features and plugins that enhance the development process without altering its functionalities. Android Studio, which is the official IDE for Android app development and built on JetBrain's IntelliJ IDEA, is an example of an IDE that supports third-party plugins developed by external vendors [25].

The ACVED plugin is designed to address the limitations of real-time code vulnerability detection methods by employing an accurate ensemble learning approach. As a result, it can be integrated into Android Studio to provide tool support for detecting vulnerabilities in real-time.

3 Development of Vulnerability Detection Model

The dataset selection process, the process of building the model and web API, how to use XAI in conjunction with prediction probabilities, and model enhancements are discussed in this section. The entire process of model development is illustrated in Fig. 1.

3.1 Dataset Selection

The first step in developing a vulnerability detection model is to select an appropriate dataset. For this purpose, the LVDAndro dataset [20], which is properly labelled based on CWE-IDs, was chosen to train the AI-based model. The LVDAndro dataset was created by combining the capabilities of MobSF and Qark vulnerability scanners, leveraging the strengths of both tools. An analysis was carried out on the LVDAndro dataset to determine its characteristics. Table 1 displays the fields that are included in the dataset. Although the processed code, vulnerability status, and CWE-IDs are essential for detecting vulnerabilities, other fields such as severity level and vulnerability category can also provide

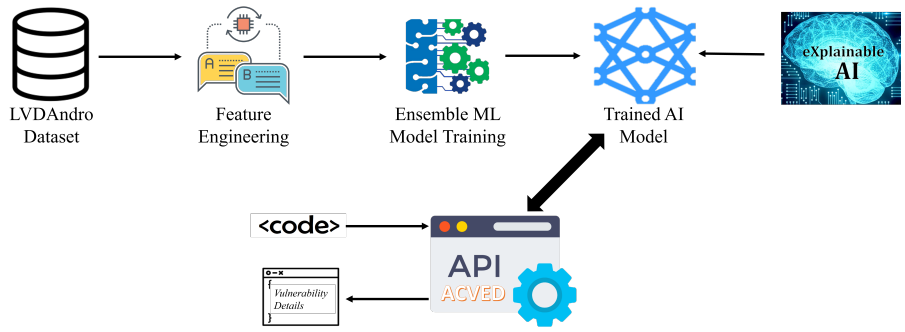

Fig. 1: Overall Model Development Process

Table 1: 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 value
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

additional predictive information. Table 2 provides statistics on the dataset, and the CWE distribution of the LVDAndro dataset is depicted in Fig. 2.

The number of non-vulnerable source code samples generally exceeds the number of vulnerable samples in the datasets generated from real applications. This data imbalance issue was addressed by down-sampling the non-vulnerable examples in the dataset. Vulnerable code examples consist of code lines for 23 CWE-IDs as shown in Fig. 2. However, for certain CWE-IDs such as 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 limited examples of vulnerable code. To handle this, a new class named *Other* was introduced and used to reassign the labels for these source code samples.

Table 2: 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

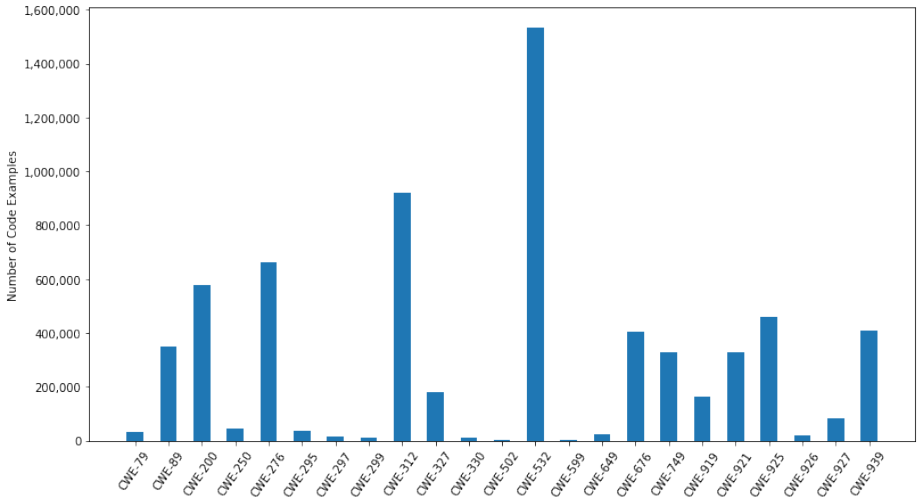

Fig. 2: CWE-ID Distribution

3.2 Model Building

When constructing the model, the LVDAndro dataset was divided into 75% for training and 25% for testing. Since the model needs to predict both the vulnerability status and the vulnerability category based on CWE, two classification tasks were performed: binary and multi-class classification as a continuation of the previous work in [18]. To create the feature vectors for these tasks, the n-gram technique with ngram_range = 1,3 and a minimum document frequency (min_df) of 100 and maximum document frequency (max_df) of 40 was used to generate two feature vectors. For the binary classification, the feature vector was created using the processed_code and vulnerability_status, while for the multi-class classification, the feature vector was created using the processed_code and CWE-ID.

To determine which classifiers perform well in both binary and CWE-based multi-class classification, widely used learning classifiers including NB, LR, DT, SVM, RF, GB, XGB, and MLP were analysed [19]. Then an ensemble learning model was built using the Stacking classifier from Scikit-learn, and the previously

analysed learning classifiers were used as estimators. The ensemble model was evaluated using five-fold cross-validation, and prediction probability, decision function, and predictions were evaluated for each estimator.

The performance of each individual classifier and the proposed ensemble model was compared based on F1-scores and accuracies for both binary and multi-class classification. The results of the comparison of accuracies and the macro averages of precision, recall, and F1-score are presented in Table 3.

Table 3: 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 3 reveals that the proposed ensemble model achieves high accuracy, precision, recall, and F1-score combinations in both binary and multi-class classifications. This result is likely due to the fact that the ensemble stacking classifier combines the capabilities of all the other classifiers. The trained ensemble model has an accuracy of 95% for both binary and multi-class classification, and 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 can be found in Table 4.

Table 4: F1-Score for each CWE-ID with Ensemble Model

CWE-ID	Precision	Recall	F1-Score	Number of Examples
CWE-89	1.00	1.00	1.00	2,036
CWE-200	0.94	0.96	0.95	5,665
CWE-276	0.97	0.98	0.97	6,382
CWE-312	0.93	0.95	0.94	7,649
CWE-532	0.98	0.99	0.99	9,254
CWE-676	1.00	1.00	1.00	2,378
CWE-749	0.65	0.90	0.76	1,898
CWE-921	0.95	0.90	0.93	1,914
CWE-925	0.99	0.99	0.99	3,392
CWE-939	0.92	0.71	0.80	2,961
Other	0.96	0.90	0.93	4,467

3.3 Web API

Two pickle files were saved for each trained model, one for the classifier and one for the vectorizer. These files correspond to the trained ensemble model for both binary and multi-class classifications. They were used as inputs to the backend of a Flask-based web API developed using Python.

The web API receives a source code line from the user via a GET request and checks it for vulnerabilities. The code line is processed using the same techniques as those used in LVDAndro [20]. Specifically, the code replaces all user-defined string values with "user_string," except for string values containing IP addresses and encryption algorithms (AES, SHA-1, and MD5) due to the potential for vulnerabilities such as CWE-200 (exposure of sensitive information to unauthorised actors) and CWE-327 (use of a broken or risky cryptographic algorithm). Additionally, all comments are replaced with "//user_comment."

Upon initialisation of the web API, the pre-trained binary and multi-class model pickle files are loaded. When a user request is received, the vectorizer from the binary classification is used to transform the processed source code line. The transformed code is then passed to the binary model to obtain the vulnerability status. If the code line is predicted to be vulnerable, the code line is transformed using the loaded multi-class classification vectorizer and then passed to the multi-class learning model to predict the CWE-ID.

3.4 Prediction Probabilities with XAI

After predicting the vulnerability status and the CWE-ID, the processed source code is passed to the Lime package in Python, which supports XAI, to obtain prediction probabilities and explanations for both binary and multi-class models. The Lime package provides the contributions of each word in the processed source code line for both vulnerability prediction and vulnerable category prediction probabilities. Finally, the prediction results are returned to the user in the form of JSON responses, as shown in Fig. 3 and Fig. 4.

The Python Lime library's *show_in_notebook* function can provide visual aids for interpreting XAI prediction probabilities. Fig. 5 demonstrates how Lime-based XAI predictions 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 associated with CWE-532, which the model correctly 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. This underscores the need for developers to exercise caution when using log statements in production-level applications, as attackers may exploit loopholes in the application by checking the log file. Encryption processes can be employed to generate log files in an encrypted form instead of plain text. Fig. 6 demonstrates how can be represented graphically if there are no any vulnerabilities in a given code line.

```
{
    "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',
    -0.00029704628779474 57)]",
    "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"
}
```


Fig. 3: API Responses Example for Vulnerable Code


```
{
    "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"
}
```


Fig. 4: API Responses Example for Non-vulnerable Code

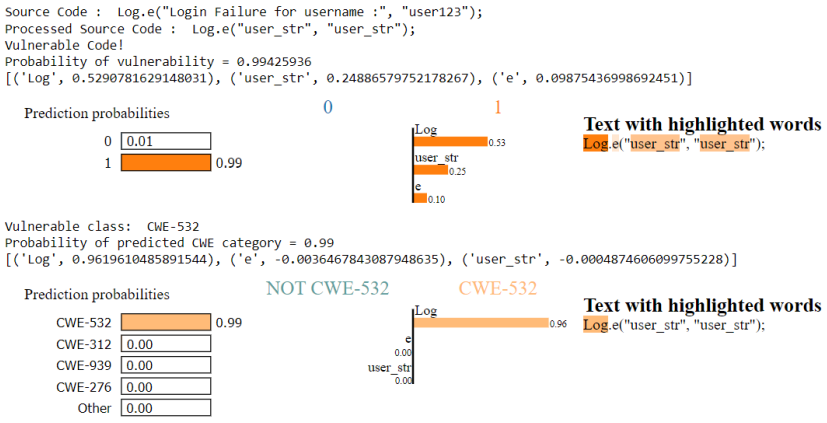

Fig. 5: Vulnerable Code

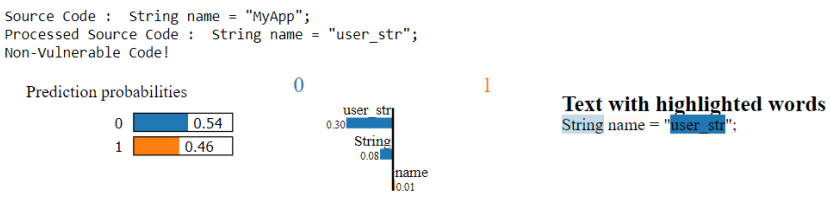

Fig. 6: Non-vulnerable Code

3.5 Continuous Model Enhancements

To keep pace with evolving threats and achieve optimal performance, the model must continuously evolve to detect new vulnerabilities and enhance its capabilities. Regular updates are essential to identify the most current source code vulnerabilities. As a result, the back-end model can improve by receiving updates to the LVDAndro training dataset.

If an update to the LVDAndro dataset is available, the model must undergo training. Otherwise, the pre-trained model can be used with the ACVED plugin. During re-training, if any classification metrics are better than those of the existing model, the new model will be employed for subsequent predictions within the plugin. The duration of model training typically relies on the available machine resources and the dataset size. In this case, the initial model was constructed using a machine equipped with an Intel Core i5 processor and 16GB of memory, taking approximately 55 minutes.

4 Application of ACVED Plugin

This section outlines how to use the ACVED plugin after integrating it with Android Studio. Furthermore, it includes a comparison of the performance of both the model and the plugin. The process of the ACVED plugin is illustrated in Fig. 7.

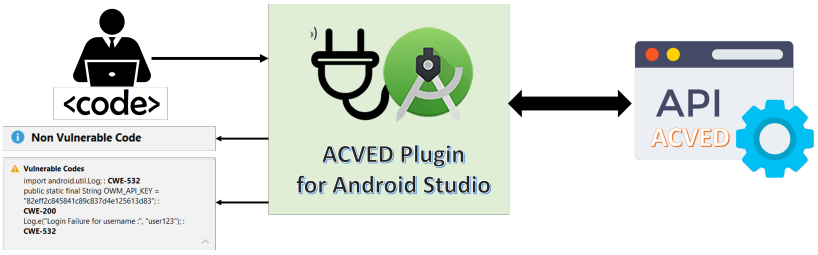

Fig. 7: Process of ACVED Plugin

4.1 Plugin Integration and Usage

The ACVED plugin, which functions as an Android Studio plugin, was created utilising IntelliJ IDEA. This plugin is capable of receiving requests (in the form of source code) from Android Studio and returning responses as notifications containing results generated by the web API.

In order to use the newly developed plugin, it must first be integrated into the Android Studio IDE. The plugin is available for download in the form of a jar file from the ACVED GitHub repository. To install the plugin into the latest version of Android Studio, simply follow the standard procedure for installing a third-party plugin. For older versions of Android Studio, the version number can be adjusted in the plugin.xml file to accommodate the appropriate version[2]. Once the plugin has been successfully installed, suggestions for resolving vulnerabilities can be retrieved as a balloon notification (see Appendix A). The ACVED plugin provides two options to detect vulnerabilities.

- **Quick Check**: Scan the whole source code file to detect the presence of vulnerable source code.
- **Detailed Check**: Detect if any vulnerability is associated with a particular code line.

When conducting a quick search, the developer will receive a balloon notification indicating whether vulnerable code is present in the source file. If no vulnerable code is found, that also will be notified. However, if no vulnerable code is detected, a notification will be displayed that specifies the vulnerable code lines and their corresponding CWE-IDs.

Upon conducting a detailed check, a notification will be received indicating the vulnerability status of the source code. If the code being focused on by the cursor is found to be vulnerable, a balloon notification will be displayed, containing a description of the vulnerability as well as a suggestion for mitigating it. The notification will also provide the binary classification prediction for vulnerability status, the associated CWE-ID, and the prediction probability for the CWE category in the multi-class classification prediction. Furthermore, the contribution of each word to the probability in both the binary and multi-class classification approaches will be indicated. The severity of the vulnerability will determine the type of notification (information or warning). To provide more detailed information about the vulnerability, ACVED offers suggestions for overcoming it by referencing the CWE repository [3].

The ability to re-perform the vulnerability check allows developers to examine how the probabilities vary when specific code lines are altered. This feature can be particularly useful in situations where 100% mitigation is not possible. For instance, in some cases, it may be necessary to maintain log file records for bug-fixing purposes, even in production-level applications.

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

4.2 Plugin Performance

The ACVED plugin's accuracy and efficiency were evaluated by benchmarking it against the MobSF and Qark scanners, which were used to construct the LVDAndro dataset. To compare the accuracy of the detection of vulnerable code for new data, a total of 2,216 source code lines were utilised. This set included 604 examples of vulnerable code lines obtained from the CWE repository and 1,612 lines of well-known non-vulnerable code from real-world applications. These code lines were integrated into an Android app project, which was then scanned using both the MobSF and Qark scanners. The same code lines were then passed to the developed API by parsing them using the Quick Check option of the ACVED plugin. The accuracy, precision, recall, and F1-Score of each tool were compared and summarised in Table 5.

Table 5: Comparison of Accuracy, Precision, Recall and F1-Score of ACVED with MobSF and Qark

Performance Metrics	**MobSF**	**Qark**	**ACVED**
Accuracy	91%	89%	**94%**
Precision	0.92	0.92	**0.94**
Recall	0.95	0.93	**0.97**
F1-Score	0.93	0.92	**0.95**

Upon comparison, ACVED outperformed MobSF and Qark in predicting vulnerabilities in unseen code samples, achieving a high accuracy rate of 94%, along with a precision of 0.94, recall of 0.97, and F1-Score of 0.95. Additionally, ACVED was able to significantly decrease the false negative rate, indicating its effectiveness in reducing potential security risks associated with its predictions. Moreover, when compared with MobSF and Qark, ACVED stands out as the sole method capable of detecting code vulnerabilities during development.

In order to compare the efficiency of vulnerability detection methods, fifty open-source Android projects were downloaded from GitHub and scanned them using the ACVED plugin (integrated with Android Studio), MobSF, and Qark. The apps were categorised based on size, with five apps per category. The average analysis times for each method were measured for each category, and the experiments were conducted on a Windows OS environment with a Core i5 processor and 16GB RAM. The results, as shown in Table 6, indicate that ACVED is faster at detecting vulnerabilities, taking only 206.1s compared to MobSF's 344.8s and Qark's 419.9s. It is worth noting that this performance comparison was conducted for completed applications due to existing vulnerability scanner limitations. However, the ACVED plugin's main strength is that it does not require building the entire application.

Table 6: Comparison of Average Time Taken to Analyse apps

App Categorise	MobSF	Qark	ACVED
$Size < 1MB$	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**

The usage of the ACVED plugin in the Android Studio IDE does not disrupt the standard coding process. Developers can continue coding in their usual way. Once a quick or detailed check has been performed, a balloon notification containing vulnerability prediction results is displayed to the developer. The results obtained from the detailed check option can be utilised to modify the code and eliminate vulnerabilities. The detailed check option also includes a clear vulnerability description, allowing developers to easily comprehend it. Additionally, by following the link provided in the notification, developers can study the vulnerability in greater depth. The ACVED plugin is capable of delivering a prediction for one code line in less than 300 milliseconds. The plugin's performance was evaluated in an Android Studio Chipmunk - 2021.2.1 version in a Windows OS environment with a Core i5 processor and 16GB RAM. However, initiating the ACVED API took between 3 to 10 seconds in the same environment before executing the ACVED plugin. As a result, app developers do not need to devote any additional time or effort to obtain these real-time prediction results.

The plugin's performance was evaluated through a survey in which 63 developers were requested to rate their satisfaction levels in various aspects including Accuracy of Prediction, Efficiency of Prediction, Ease of Integration and Configuration, Ease of Use, Usefulness of Mitigation Suggestions, Look and Feel, and Overall Satisfaction using a 5-point Likert scale. The feedback collected was represented visually in Fig. 8.

The survey results showed that the majority of app developers, 87%, were highly satisfied with the accuracy and efficiency of the predictions made by the plugin. Most developers, 89%, also found the mitigation recommendations to be useful. However, the plugin's usability, integration, and look and feel aspects received lower satisfaction ratings, with only 22% being highly satisfied with usability and integration, and 57% not being highly satisfied with the look and feel. This feedback can be used to improve the plugin's features, such as integrating mitigation suggestions in a manner similar to how syntax errors are indicated. Nevertheless, the overall satisfaction rate was high, with 79% of developers being highly satisfied and 21% satisfied. The plugin has the potential for broader use to address Android source code vulnerabilities with further development.

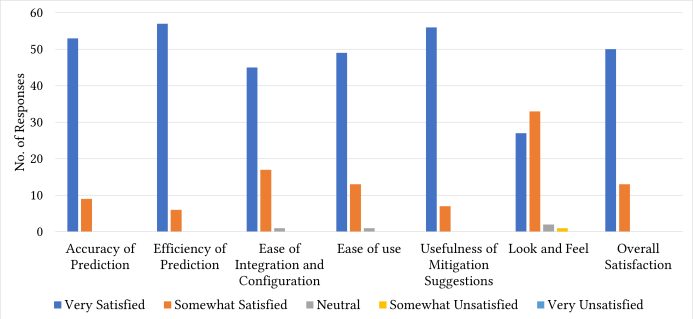

Fig. 8: Survey Results - Satisfaction of the plugin

5 Conclusion and Future Work

With numerous mobile applications available on Google Play and other Android marketplaces, it is not uncommon for developers to overlook security best practices, leaving their applications vulnerable to attacks. To bridge this gap and assist Android app developers in mitigating source code vulnerabilities in real-time, an AI-powered plugin called ACVED was introduced in this study. An ensemble learning model was trained using the LVDAndro dataset and integrated into the ACVED plugin, which is equipped with an API to detect source code vulnerabilities. The model achieved 95% accuracy in both binary classification and CWE-based multi-class classification, with F1-Scores of 0.95 and 0.93, respectively. The ACVED plugin provides XAI-based reasons for predictions to help developers quickly address vulnerabilities by considering the prediction probabilities of each word in the code line. All necessary instructions and source code for the dataset, model, API, and ACVED plugin are freely available on GitHub. In order to improve the prediction model's performance, it is feasible to incorporate expert knowledge from app developers. This integration would enable the plugin suggestions to provide regular developers with advanced security recommendations to effectively address vulnerabilities. Another potential improvement is to fine-tune the model to identify intricate vulnerability patterns supported with generative pre-trained transformer models. Additionally, one can consider utilising federated learning methods, which would allow the distribution of the model to individual entities, such as app development companies, to retrain the model while safeguarding the confidentiality of the protected code bases. By implementing this approach, it is anticipated that the number of detectable vulnerabilities of ACVED will also experience additional growth.

Acknowledgment

We thank Robert Gordon University - UK and the Accelerating Higher Education Expansion and Development grant (AHEAD) of Sri Lanka, University of Kelaniya - Sri Lanka for their support.

A Appendix

Once the plugin has been integrated (as in Fig 9), to activate the quick check feature, the user can navigate to *Tools - Check Source Vulnerability* or use the shortcut key *CTRL+ALT+E* within the Android Studio. This feature provides a rapid search for identifying vulnerabilities, notifying the developer of the specific lines of vulnerable code and their corresponding CWE-IDs as depicted in Fig. 10 and Fig. 11.

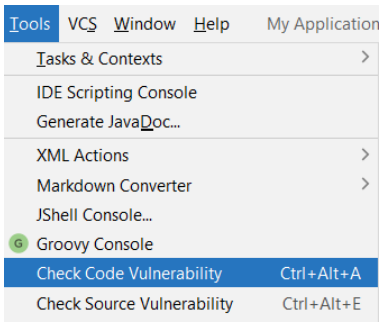

Fig. 9: Android Studio Tools Menu after Integrating ACVED

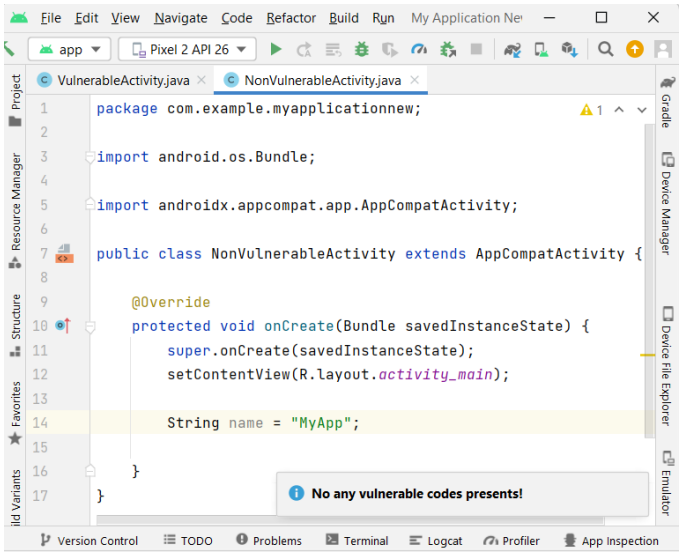

Fig. 10: Quick Check Notifications - No Any Vulnerable Code Lines

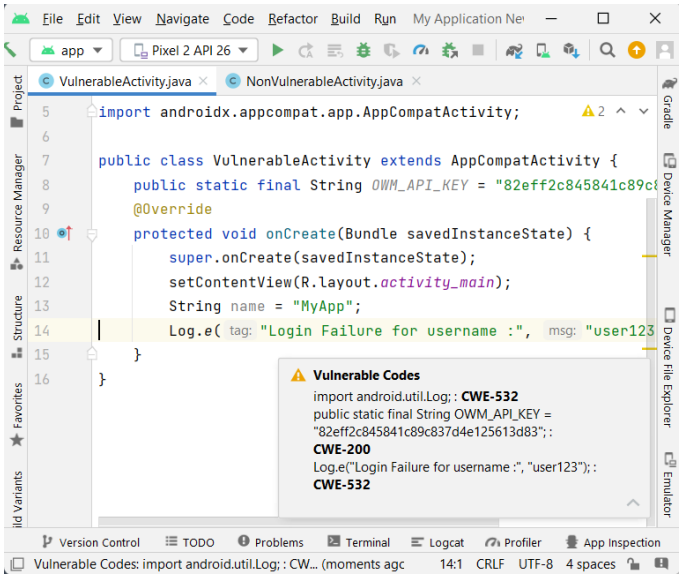

Fig. 11: Quick Check Notifications - Contain Vulnerable Code Lines

Alternatively, the detailed check feature can be activated by selecting *Tools - Check Code Vulnerability* or by using the shortcut key *CTRL+ALT+A* while the cursor is focused on a particular code line. Fig. 12 presents an example of a detailed check executed on a vulnerable code line where the cursor is positioned on the statement *Log.e("Login Failure for username :", "user123");*.

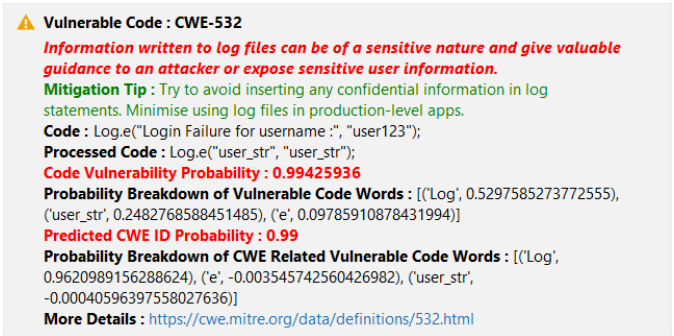

Fig. 12: Detailed Check - Balloon Notification

References

1. Albakri, A., Fatima, H., Mohammed, M., Ahmed, A., Ali, A., Ali, A., Elzein, N.M.: Survey on reverse-engineering tools for android mobile devices. Mathematical Problems in Engineering **2022** (2022). https://doi.org/10.1155/2022/4908134
2. 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, accessed: 2023-02-03
3. Corporation, M.: Common weakness enumeration (cwe) (2023), https://cwe.mitre.org/, accessed: 2023-02-01
4. Corporation, M.: Cve details (2023), https://www.cvedetails.com/, accessed: 2023-02-01
5. Gajrani, J., Tripathi, M., Laxmi, V., Somani, G., Zemmari, A., Gaur, M.S.: Vulvet: Vetting of vulnerabilities in android apps to thwart exploitation. Digital Threats: Research and Practice **1**(2), 1–25 (2020). https://doi.org/10.1145/3376121
6. Garg, S., Baliyan, N.: Android security assessment: A review, taxonomy and research gap study. Computers & Security **100**, 102087 (2021). https://doi.org/j.cose.2020.102087
7. Ghaffarian, S.M., Shahriari, H.R.: Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Comput. Surv. **50**(4) (aug 2017). https://doi.org/10.1145/3092566
8. Kouliaridis, V., Kambourakis, G.: A comprehensive survey on machine learning techniques for android malware detection. Information **12**(5), 185 (2021). https://doi.org/10.3390/info12050185
9. Krasner, H.: The cost of poor software quality in the us: A 2020 report. In: Proc. Consortium Inf. Softw. QualityTM (CISQTM) (2021)
10. Mahindru, A., Singh, P.: Dynamic permissions based android malware detection using machine learning techniques. In: Proceedings of the 10th innovations in software engineering conference. pp. 202–210 (2017). https://doi.org/10.1145/3021460.3021485
11. McDermid, J.A., Jia, Y., Porter, Z., Habli, I.: Artificial intelligence explainability: the technical and ethical dimensions. Philosophical Transactions of the Royal Society A **379**(2207), 20200363 (2021)
12. 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. p. 43–52. PROMISE, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3127005.3127010
13. Nagaria, B., Hall, T.: How software developers mitigate their errors when developing code. IEEE Transactions on Software Engineering **48**(6), 1853–1867 (2022). https://doi.org/10.1109/TSE.2020.3040554
14. 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. pp. 64–71. IBM Corp., USA (2019), https://dl.acm.org/doi/abs/10.5555/3370272.3370279
15. NIST: National vulnerability database (2023), https://nvd.nist.gov/vuln, accessed: 2023-02-21
16. Rajapaksha, S., Senanayake, J., Kalutarage, H., Al-Kadri, M.O.: Ai-powered vulnerability detection for secure source code development. In: Innovative Security

Solutions for Information Technology and Communications. pp. 275–288. Springer Nature Switzerland, Cham (2023)
17. Senanayake, J., Kalutarage, H., Al-Kadri, M.O.: Android mobile malware detection using machine learning: A systematic review. Electronics **10**(13), 1606 (2021). https://doi.org/10.3390/electronics10131606
18. Senanayake, J., Kalutarage, H., Al-Kadri, M.O., 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. p. 1255–1257. ASIA CCS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3488932.3527290
19. Senanayake, J., Kalutarage, H., Al-Kadri, M.O., Petrovski, A., Piras, L.: Android source code vulnerability detection: A systematic literature review. ACM Comput. Surv. **55**(9) (jan 2023). https://doi.org/10.1145/3556974
20. Senanayake, J., Kalutarage, H., Al-Kadri, M.O., Petrovski, A., Piras, L.: 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), accepted
21. Shezan, F.H., Afroze, S.F., Iqbal, A.: Vulnerability detection in recent android apps: An empirical study. In: 2017 International Conference on Networking, Systems and Security (NSysS). pp. 55–63. IEEE, Dhaka, Bangladesh (2017). https://doi.org/10.1109/NSysS.2017.7885802
22. Srivastava, G., Jhaveri, R.H., Bhattacharya, S., Pandya, S., Rajeswari, Maddikunta, P.K.R., Yenduri, G., Hall, J.G., Alazab, M., Gadekallu, T.R.: Xai for cybersecurity: State of the art, challenges, open issues and future directions (2022). https://doi.org/10.48550/ARXIV.2206.03585
23. Statcounter: Mobile operating system market share worldwide (2023), https://gs.statcounter.com/os-market-share/mobile/worldwide/, accessed: 2023-04-01
24. Statista: Average number of new android app releases via google play per month from march 2019 to march 2023 (2023), https://www.statista.com/statistics/1020956/android-app-releases-worldwide/, accessed: 2022-04-03
25. 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 **25**(5), 589–603 (2020). https://doi.org/10.26599/TST.2019.9010067
26. Thomas, G., Devi, A.: A study and overview of the mobile app development industry. International Journal of Applied Engineering and Management Letters pp. 115–130 (2021). https://doi.org/10.5281/zenodo.4966320.
27. de Vicente Mohino, J., Bermejo Higuera, J., Bermejo Higuera, J.R., Sicilia Montalvo, J.A.: The application of a new secure software development life cycle (s-sdlc) with agile methodologies. Electronics **8**(11) (2019). https://doi.org/10.3390/electronics8111218
28. 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). pp. 569–574. IEEE (2017). https://doi.org/10.1109/ICISCE.2017.124