

Labelled Vulnerability Dataset on Android Source Code (LVDAndro) to Develop AI-Based Code Vulnerability Detection Models

Janaka Senanayake^{1,2}^a, Harsha Kalutarage¹^b, Mhd Omar Al-Kadri³^c, Luca Piras⁴^d
and Andrei Petrovski¹^e

¹*School of Computing, Robert Gordon University, Aberdeen, U.K.*

²*Faculty of Science, University of Kelaniya, Kelaniya, Sri Lanka*

³*School of Computing and Digital Technology, Birmingham City University, Birmingham, U.K.*

⁴*Department of Computer Science, Middlesex University, London, U.K.*


Keywords: Android Application Security, Code Vulnerability, Labelled Dataset, Artificial Intelligence, Auto Machine Learning.


Abstract: Ensuring the security of Android applications is a vital and intricate aspect requiring careful consideration during development. Unfortunately, many apps are published without sufficient security measures, possibly due to a lack of early vulnerability identification. One possible solution is to employ machine learning models trained on a labelled dataset, but currently, available datasets are suboptimal. This study creates a sequence of datasets of Android source code vulnerabilities, named LVDAndro, labelled based on Common Weakness Enumeration (CWE). Three datasets were generated through app scanning by altering the number of apps and their sources. The LVDAndro, includes over 2,000,000 unique code samples, obtained by scanning over 15,000 apps. The AutoML technique was then applied to each dataset, as a proof of concept to evaluate the applicability of LVDAndro, in detecting vulnerable source code using machine learning. The AutoML model, trained on the dataset, achieved accuracy of 94% and F1-Score of 0.94 in binary classification, and accuracy of 94% and F1-Score of 0.93 in CWE-based multi-class classification. The LVDAndro dataset is publicly available, and continues to expand as more apps are scanned and added to the dataset regularly. The LVDAndro GitHub Repository also includes the source code for dataset generation, and model training.


1 INTRODUCTION


Approximately 90,000 Android mobile apps are released through the Google Play Store monthly. In January 2023, Android holds a 71.74% market share (Statista, 2023; Statcounter, 2023). However, many of these apps are developed without adhering to secure coding best practices and standards, resulting in source code vulnerabilities, which appeal to attackers. In contrast to iOS, Android applications are not thoroughly checked for security aspects (Senanayake et al., 2021), and therefore the security of these apps is not guaranteed, and they may fail to comply with rigorous security protocols.


To ensure the security of apps, it is recommended to implement secure coding practices while writing the code, as many vulnerabilities stem from flaws in the source code. The Security Development Lifecycle (SDL) recommends following secure development practices in real-time, rather than waiting until the application is developed (Souppaya et al., 2021). To help enforce these practices, researchers have developed automated tools for identifying Android app vulnerabilities using various scanning methods, including conventional, Machine Learning (ML), and Deep Learning (DL) methods (Shezan et al., 2017; Senanayake et al., 2023). These methods utilise three analysis approaches: static, dynamic, and hybrid. However, many existing vulnerability detection methods require Android Application Package (APK) files that are ready to be installed, limiting their usefulness during development. To overcome that, it is possible to use well-trained ML/DL models, which can detect vulnerabilities simultaneously when the code is writ-

^a  <https://orcid.org/0000-0003-2278-8671>

^b  <https://orcid.org/0000-0001-6430-9558>

^c  <https://orcid.org/0000-0002-1146-1860>

^d  <https://orcid.org/0000-0002-7530-4119>

^e  <https://orcid.org/0000-0002-0987-2791>

ten. A properly labelled dataset on Android source code vulnerability is required to train such models. Hence, this paper makes the following contributions.

1. Producing a properly labelled novel dataset of Android source code vulnerabilities named LVDAndroid, which offers the following characteristics:
 - A. LVDAndroid contains more than fifteen million distinct code samples scanned from over fifteen thousand Android Apps;
 - B. within LVDAndroid, vulnerable code examples were labelled with Common Weakness Enumeration (CWE)¹ identifications and contain additional attributes such as vulnerability category, severity, and description. As a result, this dataset is unique compared to existing ones;
 - C. within LVDAndroid, the labelling process was done by combining multiple vulnerability scanners including Mobile Security Framework (MobSF)² and Quick Android Review Kit (Qark)³. Hence ML models trained with LVDAndroid learn the capabilities of all scanners.
2. Performing binary and multi-class classification-related AutoML experiments as a Proof-of-Concept (PoC) to determine the applicability of the LVDAndroid dataset for Android source code vulnerability detection with ML. The classifiers achieved:
 - A. accuracy of 94% and F1-Score of 0.94 in binary classification, which predicts vulnerable codes;
 - B. accuracy of 94% and F1-Score of 0.93 in multi-class classification, which predicts the CWE ID of a vulnerable code.
3. Making the dataset available for public access as a GitHub repository⁴, along with the dataset generation scripts and the instructions to enhance the dataset by adding more data as needed.

The remaining sections of the paper are structured as follows: in Section 2, prior research on the subject is reviewed, and in Section 3, the dataset generation is discussed. Section 4 outlines the attributes and the statistics of the LVDAndroid dataset, and Section 5 examines how the LVDAndroid can be used to train ML models to identify vulnerabilities in Android code. Section 6 provides the conclusion by discussing the findings and future plans.

¹<https://cwe.mitre.org/>

²<https://github.com/MobSF/Mobile-Security-Framework-MobSF>

³<https://github.com/linkedin/qark/>

⁴<https://github.com/softwaresec-labs/LVDAndroid>

2 RELATED WORK

This section discusses the related studies on code vulnerabilities and datasets, which can be used to detect software vulnerabilities using ML-based methods.

Organisations and communities have identified a number of vulnerabilities. CWE and Common Vulnerabilities and Exposures (CVE)⁵ are generally used as references for identifying weaknesses and vulnerabilities across many programming languages. As a result, mobile app developers can also refer to these references to identify vulnerabilities and address security issues in their source code.

Previous research has proposed a number of datasets and repositories focused on vulnerabilities. For instance, AndroVul (Namrud et al., 2019) is a repository that deals with security issues related to Android, such as security code smells, dangerous permissions, and dangerous shell commands. It was created by analysing APKs downloaded from AndroZoo (Allix et al., 2016), and serves as a benchmark for detecting Android malware. It can also be used for ML experiments to detect malware with static analysis. Another dataset, introduced in (Challande et al., 2022), is a commit-level dataset for real-world vulnerabilities, which has analysed more than 1,800 projects and over 1,900 vulnerabilities based on CVE from the Android Open Source Project. Ghera (Mitra and Ranganath, 2017), an open-source repository of benchmarks, has captured 25 known vulnerabilities in Android apps and also presented some common characteristics of vulnerability benchmarks and repositories. Additionally, the National Vulnerability Database (NVD)⁶ is another dataset that can be utilised as a reference for vulnerabilities. However, it lacks the ability to support the development of AI-based models because vulnerable categories for code lines are not correctly labeled.

It is possible to create new datasets by examining Android apps for vulnerabilities. There are two methods for analysing Android applications. The first approach involves reverse-engineering the developed APKs and analysing the code. However, this method requires a pre-built application and is not applicable in the early stages of the SDLC (Senanayake et al., 2022). The second approach involves analysing the source code while it is being written. Both methods utilise static, dynamic, and hybrid analysis techniques as the initial step of application scanning. Static analysis techniques identify code issues without executing the application or source code and can be categorised into two types: manifest analysis and

⁵<https://cve.mitre.org/>

⁶<https://nvd.nist.gov/vuln>

code analysis. Manifest analysis can identify package names, permissions, activities, services, intents, and providers. On the other hand, code analysis provides deeper insights into the source code by analysing features such as API calls, information flow, native code, taint tracking, clear-text analysis, and opcodes (Senanayake et al., 2021). Dynamic analysis, in contrast to static analysis, requires a runtime environment to execute the application for scanning. This approach is commonly used for malware detection and identifying vulnerabilities in pre-built applications. Hybrid analysis combines both static and dynamic analysis techniques, where static analysis is used to analyse the manifest file and source code files, and dynamic analysis is used to analyse the application's characteristics at runtime.

Various tools are available to conduct such analysis. For instance, APKTool⁷ can extract code-level information by decompiling the APK using a static analyser. This tool is widely used as the foundation for vulnerability detection methods based on static analysis that involve reverse-engineering APKs (Senanayake et al., 2021). Qark is another tool that can identify vulnerabilities in Android apps by examining pre-built APKs or source code files. MobSF, on the other hand, uses a hybrid analysis model to detect vulnerabilities, malware, and perform penetration testing. It is a security framework designed for Android and iOS, which offers REST API for development integration. HornDroid (Calzavara et al., 2016) is a tool that can analyse information flow in Android apps by abstracting their semantics to construct security properties, while COVERT (Bagheri et al., 2015) can perform compositional analysis of inter-app vulnerabilities in Android. Another tool that can be useful for identifying vulnerabilities in Android source code through static analysis is Android Lint (Goaër, 2020), which uses Abstract Syntax Trees (AST) or Universal AST generated from source code.

Previous research has noted that both ML-based and non-ML-based methods can be employed to detect vulnerabilities. However, in recent years, there has been a greater tendency to use ML-based approaches over non-ML-based methods (Ghaffarian and Shahriari, 2017). Additionally, model accuracy and performance can be improved by enhancing datasets and tuning parameters through various ML experiments. While Alloy (Bagheri et al., 2018) and VulArcher (Qin et al., 2020) have presented non-ML-based techniques such as formal and heuristic-based methods, ML-based vulnerability detection methods have been proposed in studies such as (Senanayake et al., 2022; Gajrani et al., 2020). These studies have

employed various classifiers, including Decision Tree (DT), Naive Bayes (NB), AdaBoost (AB), Random Forest (RF), Gradient Boosting (GB), Extreme Gradient Boosting (XGB), Logistic Regression (LR), Support Vector Classifier (SVC), and Multi-Layer Perception (MLP) trained on labelled datasets.

The detection of source code vulnerabilities in Android has been a challenge due to the absence of an accurate method during code writing, as well as a lack of appropriately labelled datasets to train machine learning (ML) models for vulnerability prediction (Senanayake et al., 2023). To address this gap, the LVDAndro dataset is introduced in this study, and a proof of concept (PoC) is presented that uses ML techniques to detect Android code vulnerabilities.

3 DATASET GENERATION PROCESS

The LVDAndro dataset is a comprehensive and diverse collection of labelled data that is specifically designed to address the challenges of detecting Android source code vulnerabilities using ML techniques. The dataset contains a wide range of source code samples with varying degrees of complexity and security vulnerabilities. The overall process of LVDAndro dataset generation is illustrated in Figure 1, and the generation process consists of three main stages, as follows.

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

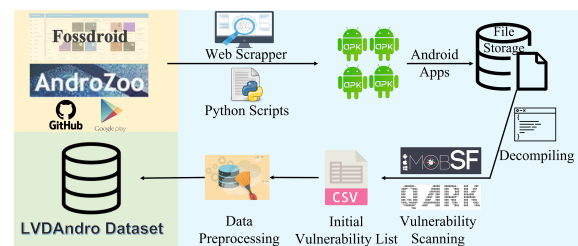


Figure 1: LVDAndro dataset generation process.

3.1 Scrapping APKs and Source Files (Data Collection)

To generate the LVDAndro dataset, the first step is to scrape APKs, and their source code, from application repositories. This includes Google Play, Fossdroid

⁷<https://ibotpeaches.github.io/Apktool/>

(Simonin, 2023), AndroZoo (Allix et al., 2016), and some well-known malware repositories (Senanayake et al., 2021). Python scripts were used to download APKs and their source code from GitHub repositories. An experiment was also carried out to investigate whether source code from reverse-engineered APKs could be used to generate the dataset instead of relying on the original source code. This was due to the lack of open-source APKs and the high availability of closed-source APKs. Figure 2 illustrates the sources of the downloaded apps in the current version of the dataset, which will be increased in future versions.

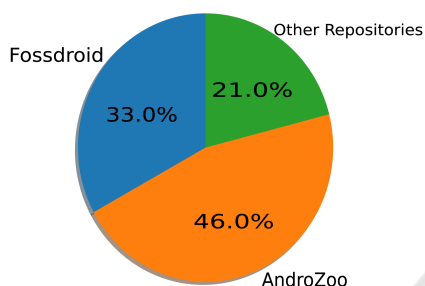


Figure 2: Source of downloaded apps.

3.2 Scanning APKs for Vulnerabilities (Data Labelling)

LVDAndro was developed to leverage ML to identify source code vulnerabilities in real-time. To create robust and effective machine learning models, a diverse dataset of APKs and source files was necessary. To achieve this, various scanning methods were employed as the second step in dataset generation, which involved scanning both the APKs and source files for vulnerabilities. This enabled the ML models to be trained on a comprehensive range of vulnerabilities.

The LVDAndro dataset was developed using the code analysis approach in the static analysis method by scanning APKs and Android project files (which include source code and file structure). Vulnerability scanning tools such as MobSF and Qark were used for this purpose. During the scanning process, these tools could identify the vulnerable lines of code and the corresponding CWE-IDs. The idea was that by using the resulting dataset to train machine learning models, the models would be able to learn from the capabilities of both scanners and perform better than either tool alone in terms of detection. A Python script was developed to automate the scanning process, and all applications were scanned using this script.

To analyse an APK or Android project using MobSF, it needs to be set up as a server, and several API requests can be made, including upload, scan, and download. When an APK or project is uploaded,

MobSF decompiles it using tools such as JADX, dex2-jar, JD-GUI. The decompiled source code or project files are then scanned for vulnerabilities. After the scan is complete, the results are stored in a local database, which is mapped to a generated hash value. The results are retrieved as a JSON object and passed to the automation Python script using the hash value. The JSON object contains details of the upload files, including vulnerability status, manifest analysis details, code analysis details, and associated files. A separate Python script is then used to extract the necessary details and the source code lines of both vulnerable and non-vulnerable codes labelled by MobSF.

To perform analysis with Qark, it is needed to run Qark as a shell script, because it does not offer APIs like MobSF does. APK of the project source file directory should be passed as parameters when running the Qark. When an APK is passed, Qark decompiles it using tools such as Fernflower, Procyon, and CFR, and then scans it to identify vulnerable lines of code. If a file is submitted, it is directly scanned. After identifying the vulnerable code lines, a Python script labels and stores them, along with a description from the scanner, vulnerability type, and severity level.

Python scripts were created to scan APKs and source files, utilising a unified approach that integrates the functionalities of MobSF and Qark. These scripts employ techniques for scanning and identifying any potential vulnerabilities in the application or source code, and the results are tagged with CWE IDs to provide relevant information.

3.3 Generating Processed Dataset (Preprocessing)

During this stage, various preprocessing steps were carried out. First, user-defined string values were replaced with *user_str*, since typical user-defined string values do not make a significant impact for vulnerabilities (Hanif and Maffeis, 2022). However, string values that included IP addresses and encryption algorithms like AES, SHA-1, and MD5 were not replaced since they may cause for vulnerabilities such as CWE-200 and CWE-327, which involve exposing sensitive information to unauthorised parties and using insecure cryptographic algorithms. Next, all comments were replaced with *//user_comment* since the language compilers ignore comments. Finally, duplicates were removed based on the processed code and the vulnerability status.

4 RESULTING DATASET

A sequence of datasets was created when generating the LVDAndro dataset. This section discusses the characteristics of these resulting datasets.

4.1 Different Datasets

The LVDAndro datasets were produced by scanning real-world Android applications. Dataset 01 was compiled by including all the popular open-source APKs and their related Android projects in FossDroid, leading to a total of 511 Apps. Another dataset, named Dataset 02, was composed of 5,503 open-source APKs and their associated projects, scanned from all the listed apps in FossDroid across 17 different categories such as Internet, Systems, Games, and Multimedia. Furthermore, Dataset 03 was formed by scanning 15,021 APKs from FossDroid, AndroVul, and Android malware repositories. This dataset includes scanned source code from both open-source and closed-source applications, consisting of 23 different CWE ID labels. If AI models need to be developed based on the types of apps, the three variations of datasets can be used. However, if there is no such requirement, Dataset 3 can be used to perform an extensive analysis and build more accurate models since it contains a large number of labelled source code examples. A summary of the LVDAndro datasets is provided in Table 1.

After processing Dataset 01 and Dataset 02, nine sub-datasets were created. These sub-datasets were generated using three different scanning approaches: MobSF scanner, Qark scanner, and the proposed combined scanner, to compare their effectiveness. Within each method, three sub-datasets were generated using only APKs, only Android projects, and both APKs and Android projects. Dataset 03, which was generated using only APKs and the combined approach, produced one dataset.

Figure 3 displays the distribution of vulnerable and non-vulnerable code samples across the datasets in LVDAndro. Observing the data, it is evident that the count of non-vulnerable source code samples is generally greater than the vulnerable source code sample count. Since the datasets were created using actual applications, it is possible that they contain a significant proportion of non-vulnerable code.

4.2 Statistics of Datasets

Table 2 presents the fields included in the LVDAndro dataset. While the processed code, vulnerability status, and CWE-IDs are necessary for detecting vul-

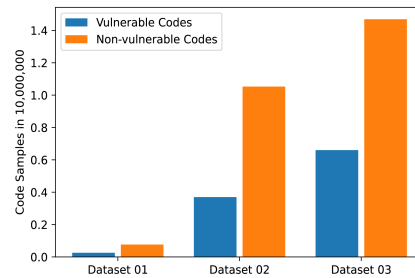


Figure 3: Vulnerable and non-vulnerable code samples distribution in each dataset (in APK Combined Approach).

nerabilities, other fields can also provide additional information for prediction.

Table 3 classifies the CWE-IDs based on their likelihood of exploitation, and Figure 4 shows the distribution of CWE-IDs in LVDAndro Dataset 03. CWE-532 has a large number of code examples, as it is common to write information to log files for debugging purposes. However, these logs may also contain sensitive information, which can be accidentally written by the developer. CWE-312 also has a significant number of code examples, as many developers tend to write sensitive information in cleartext. Most of the other CWE categories have an even distribution of code examples, whereas categories like CWE-299, CWE-502, and CWE-599 have fewer examples due to their complexity and difficulty in finding relevant instances in the context of Android source code.

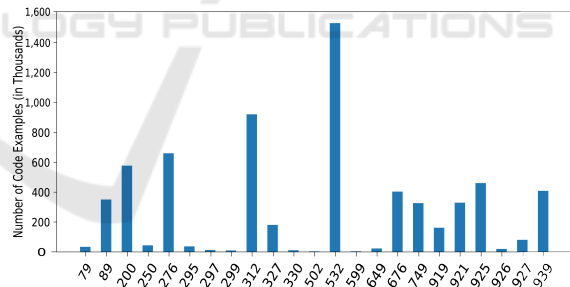


Figure 4: CWE-ID distribution in dataset 03.

Figure 5 depicts the distribution of CWE-IDs in Dataset 03 based on their CWE likelihood of exploitation values. As the dataset consists of 95% vulnerable code examples for both high and medium exploitable CWE-IDs, it is expected to be highly effective in detecting vulnerabilities.

5 DATASET USAGE

This section outlines the proof-of-concept concerning utilising LVDAndro to train machine learning models,

Table 1: Summary of the LVDAndro datasets.

Dataset	Created Date	No. of Code Samples	No. of Vulnerable Codes	No. of Non Vulnerable Codes	Vul : Non-vul Ratio	No. of CWE-IDs	Description
Dataset 01	Mar-2022	1,020,134	765,101	255,034	1:3	22	Created using 511 open-source apps. 9 sub-datasets - scanned with MobSF, Qark and Combined scanners (3 by scanning only APKs, 3 by scanning only source files, 3 by scanning both APKs and source files).
Dataset 02	Jun-2022	14,228,925	10,529,405	3,699,521	7:9	23	Created using 5,503 open-source apps. 9 sub-datasets - scanned with MobSF, Qark and Combined scanners (3 by scanning only APKs, 3 by scanning only source files, 3 by scanning both APKs and source files).
Dataset 03	Dec-2022	21,289,029	14,689,432	6,599,597	9:11	23	Created using 15,021 apps. 1 dataset - scanned with combined scanner using both open-source and closed-source apps from various sources

Table 2: Fields in LVDAndro.

Field Name	Description
Index	Auto-generated identifier
Code	Original source code line
Pprocessed_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

Table 3: Available CWE-IDs in LVDAndro.

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

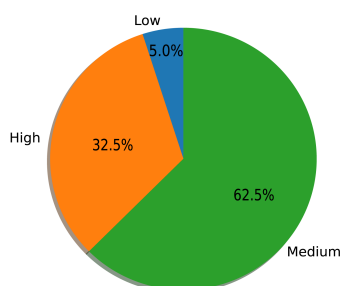


Figure 5: CWE distribution based on the likelihood of exploit.

for detecting vulnerabilities in Android source code. It shows that by training AutoML model on the LVDAndro dataset, it is possible to accurately detect and classify different types of vulnerabilities in Android source code.

5.1 Training AutoML Models

In this section, the performance of AutoML models trained on LVDAndro datasets for detecting vulnerable code lines (binary classification), and for detecting CWE-IDs (multi-class classification), are compared. To handle the data imbalance issue, the data were re-sampled, and the dataset was split into an 80:20 ratio for training and testing. The resulting performance metrics are presented in Table 4 and Table 5 for binary and multi-class classification, respectively, and are categorised by dataset.

Table 4: Performance comparison of AutoML models in binary classification.

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

According to Table 4 and Table 5, it can be observed that the combined approach yielded better results when using APKs, source files, and both for models in Dataset 01. As a result, only the combined approach was used to train AutoML models in Dataset 02. When training with Dataset 02, it was discovered that the APKs combined approach outperformed the other source combined and all combined

Table 5: Performance comparison of AutoML models in multi-class classification.

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

approaches. Therefore, only APKs were used for scanning in Dataset 03. Furthermore, using multiple sources to download APKs could potentially reduce bias and impact the overall performance. Increasing the dataset size resulted in a continuous improvement in F1-Scores for both binary and multi-class classifications. Minimising false positives and false negatives is crucial to enhance the efficiency of any ML-based solution, with minimising false negatives being more critical in this problem. To accomplish this, several measures, such as improving data quality during preprocessing and training, were taken to reduce both types of false alarms.

5.2 AutoML Model Comparison

An API was developed to detect vulnerable code lines, and the associated CWE-ID using an AutoML model trained on LVDAndro dataset. The API requires source code lines as input, and in this experiment, it was tested on a set of 3,312 source code lines (unseen data) comprising both vulnerable examples from the CWE repository and non-vulnerable examples from real applications. Subsequently, an APK was created by incorporating the same set of 3,312 source code lines, and the APK was scanned using MobSF and Qark Scanners. The accuracies of the three approaches are reported in Table 6.

Table 6: Accuracy comparison of proposed ML model with MobSF and Qark.

Approach	Accuracy
MobSF	91%
Qark	89%
Proposed Approach	94%

The detection techniques of MobSF and Qark rely on signatures, which are known for producing a high number of false negatives, while maintaining accuracy in terms of true positives. To overcome this limitation, the proposed ML-based technique trained on LVDAndro can be applied, as it has learned from the strengths of both scanners to make it more robust. With an accuracy of 94%, the proposed ML model could accurately predict the vulnerability associated with tested code. This test was performed using unseen source code, demonstrating that the proposed method can detect vulnerabilities in new APKs with high accuracy, confirming the hypothesis (PoC). By incorporating additional scanners into the pipeline and expanding the dataset with regular updates that include data related to novel vulnerabilities, the proposed method’s accuracy can be further improved. Additionally, the size and quality of the labelled dataset can also be increased.

6 CONCLUSION AND FUTURE WORKS

When developing Android mobile applications, it is essential to adopt security-focused practices, from early stages, during the overall development cycle, and it is important to receive valuable automated tool support. One way to support app developers, in identifying source code vulnerabilities, is by applying AI methods. This study presents a dataset called LVDAndro, which contains over 20 million distinct source code samples, labelled based on CWE-IDs, for identifying Android source code vulnerabilities. The dataset can be used to train machine learning models to predict vulnerabilities, achieving 94% accuracy in binary and multi-class classification, with 0.94 and 0.93 F1-Scores, respectively. The dataset is available on GitHub and ongoing efforts are underway to expand it and increase sample sizes for deeper learning models. The addition of more scanners can further increase the model's accuracy. Adopting security-focused practices and receiving automated tool support is important for developing secure Android apps.

REFERENCES

- Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2016). Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 468–471, New York, NY, USA. ACM.
- Bagheri, H., Kang, E., Malek, S., and Jackson, D. (2018). A formal approach for detection of security flaws in the android permission system. *Formal Aspects of Computing*, 30(5):525–544.
- Bagheri, H., Sadeghi, A., Garcia, J., and Malek, S. (2015). Covert: Compositional analysis of android inter-app permission leakage. *IEEE transactions on Software Engineering*, 41(9):866–886.
- Calzavara, S., Grishchenko, I., and Maffei, M. (2016). Horndroid: Practical and sound static analysis of android applications by smt solving. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 47–62, Saarbruecken, Germany. IEEE.
- Challande, A., David, R., and Renault, G. (2022). Building a commit-level dataset of real-world vulnerabilities. In *Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy*, CO-DASPY '22, page 101–106, New York, USA. ACM.
- Gajrani, J., Tripathi, M., Laxmi, V., Somani, G., Zemmari, A., and Gaur, M. S. (2020). Vulvet: Vetting of vulnerabilities in android apps to thwart exploitation. *Digital Threats: Research and Practice*, 1(2):1–25.
- Ghaffarian, S. M. and Shahriari, H. R. (2017). Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv.*, 50(4).
- Goaër, O. L. (2020). Enforcing green code with android lint. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops*, ASE '20, page 85–90, New York, NY, USA. ACM.
- Hanif, H. and Maffei, S. (2022). Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8.
- Mitra, J. and Ranganath, V.-P. (2017). 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, page 43–52, New York, NY, USA. ACM.
- Namrud, Z., Kpodjedo, S., and Talhi, C. (2019). Androvul: A repository for android security vulnerabilities. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 64–71, USA. IBM Corp.
- Qin, J., Zhang, H., Guo, J., Wang, S., Wen, Q., and Shi, Y. (2020). Vulnerability detection on android apps-inspired by case study on vulnerability related with web functions. *IEEE Access*, 8:106437–106451.
- Senanayake, J., Kalutarage, H., and Al-Kadri, M. O. (2021). Android mobile malware detection using machine learning: A systematic review. *Electronics*, 10(13):1606.
- Senanayake, J., Kalutarage, H., Al-Kadri, M. O., Petrovski, A., and Piras, L. (2022). Developing secured android applications by mitigating code vulnerabilities with machine learning. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, page 1255–1257, New York, NY, USA. ACM.
- Senanayake, J., Kalutarage, H., Al-Kadri, M. O., Petrovski, A., and Piras, L. (2023). Android source code vulnerability detection: A systematic literature review. *ACM Comput. Surv.*, 55(9).
- Shezan, F. H., Afroze, S. F., and Iqbal, A. (2017). Vulnerability detection in recent android apps: An empirical study. In *2017 International Conference on Networking, Systems and Security (NSysS)*, pages 55–63, Dhaka, Bangladesh. IEEE.
- Simonin, D. (2023). Fossdroid. <https://fossdroid.com/>. Accessed: 2023-01-02.
- Souppaya, M., Scarfone, K., and Dodson, D. (2021). Secure software development framework: Mitigating the risk of software vulnerabilities. Technical report, NIST.
- Statcounter (2023). Mobile operating system market share worldwide. <https://gs.statcounter.com/os-market-share/mobile/worldwide/>. Accessed: 2023-01-02.
- Statista (2023). Average number of new android app releases via google play per month. <https://www.statista.com/statistics/1020956/android-app-releases-worldwide/>. Accessed: 2023-02-02.