information

Article

AndroShield: Automated Android Applications Vulnerability Detection, a Hybrid Static and Dynamic Analysis Approach

Amr Amin [1,†], Amgad Eldessouki [1,†], Menna Tullah Magdy [1,†], Nouran Abdeen [1,†], Hanan Hindy [1,2,*] and Islam Hegazy [1,*]

1. Faculty of Computer & Information Sciences, Ain Shams University, Cairo 11566, Egypt; amr.amin.stdsw@cis.asu.edu.eg (A.A.); amgaddessouki@gmail.com (A.E.); menna.tullah.magdy731@gmail.com (M.T.M.); nouran.m.abdeen@gmail.com (N.A.)
2. Division of Cyber-Security, Abertay University, Dundee DD1 1HG, UK

* Correspondence: 1704847@abertay.ac.uk (H.H.); islheg@cis.asu.edu.eg (I.H.)

† These authors contributed equally to this work.

Received: 30 July 2019; Accepted: 18 October 2019; Published: 22 October 2019

Abstract: The security of mobile applications has become a major research field which is associated with a lot of challenges. The high rate of developing mobile applications has resulted in less secure applications. This is due to what is called the "rush to release" as defined by Ponemon Institute. Security testing—which is considered one of the main phases of the development life cycle—is either not performed or given minimal time; hence, there is a need for security testing automation. One of the techniques used is *Automated Vulnerability Detection*. Vulnerability detection is one of the security tests that aims at pinpointing potential security leaks. Fixing those leaks results in protecting smart-phones and tablet mobile device users against attacks. This paper focuses on building a hybrid approach of static and dynamic analysis for detecting the vulnerabilities of Android applications. This approach is capsuled in a usable platform (web application) to make it easy to use for both public users and professional developers. Static analysis, on one hand, performs code analysis. It does not require running the application to detect vulnerabilities. Dynamic analysis, on the other hand, detects the vulnerabilities that are dependent on the run-time behaviour of the application and cannot be detected using static analysis. The model is evaluated against different applications with different security vulnerabilities. Compared with other detection platforms, our model detects information leaks as well as insecure network requests alongside other commonly detected flaws that harm users' privacy. The code is available through a GitHub repository for public contribution.

Keywords: vulnerability detection; Android applications; static analysis; dynamic analysis; mobile security; user privacy

1. Introduction

The dependence on mobile applications has increased dramatically over the past decade. Statista [1] shows that in 2017 there were 178.1 billion Android application downloads which rose to 205.4 billion in 2018. It is expected to reach 258.2 billion in 2022. Moreover, the number of available applications on Google Play store escalated from around 2.3 million in the first quarter of 2016 to 3.8 million in 2018. There are—on average—6140 applications released per day. With the increasing rate of the production of Android applications, the testing phase takes less time and security testing is sometimes neglected. Android, as one of the most popular mobile open-source operating systems, suffers more from different vulnerabilities. As a result, Android applications are vulnerable to malicious users and hackers who may gain access to unauthorised information. In its report about mobile security [2], the Ponemon Institute states that around 33% of mobile applications are not tested and 38% of those involved in their study do not perform a vulnerability scan for their mobile applications.

Third-party testing is one of the suggested solutions to overcome the limited time given to the security testing for any released application; however, third parties cannot cope with the pace of the released applications. Therefore, there is a need to develop automated tools to detect vulnerabilities.

In this paper, we address the problem by introducing an automated vulnerability detection model. This model builds both static and dynamic analysis methods. Static analysis detects offline vulnerabilities (the ones that do not require the application to be running). It is based on application reverse engineering and code scanning. Dynamic analysis, on the contrary, detects the vulnerabilities that cannot be detected by static analysis. It is based on the run-time behaviour of the application. In order to perform dynamic analysis, application simulation runs are performed. After the analysis is finished, a report is generated consisting of three sections; (a) **APK Information**, which provides unique information about the uploaded application. (b) **APK Permissions**, which provides a list of the permissions granted to the app. (c) Finally, **Vulnerabilities**, which explains the detected vulnerabilities. The proposed system detects the following vulnerabilities: "Information Leaks", "Intent Crashes", "Insecure Network Requests", "Exported Android Components", "Enabled Backup Mode" and "Enabled Debug Mode". Each vulnerability is associated with a risk level and a description.

The current platform targets Android application vulnerability detection; however, it can be extended to involve other platforms. Extension to other platforms would involve overriding the emulator sniffing and code reverse engineering processes. The back-end web application and the reporting module would not require changes. The model is targeted for professional developers, on one hand, and daily application users on the other hand. The generated report includes information that the normal user can understand and can decide whether or not the application he/she is using is vulnerable. In addition, the professional developers and organisations can benefit from the details in the generated report to tackle and fix their security flaws.

The rest of the paper is organised as follows. Section 2 defines the important concepts of static and dynamic analysis. Section 3 discusses the available vulnerability detection tools and how our work fits in the current research. Section 4 visualises the proposed architecture of the hybrid model as well as the methodology. Section 5 discusses the experimental results. Finally, Section 6 summarises the paper stating the conclusion and future work.

2. Background

2.1. Vulnerability Detection Techniques

There are two techniques for automating vulnerability detection for mobile applications, static and dynamic analysis. Static analysis detects vulnerabilities without executing the application while dynamic analysis detects the vulnerabilities during the run-time of the application. Combining both techniques gives a more robust output and a higher probability of detecting malicious and vulnerable applications.

2.1.1. Static Analysis

Static analysis generally involves taking the source code—or object code in some cases—of a program and examining it without execution. The results are generated by checking the code structure, the sequences of statements, and how variable values are processed throughout the different function calls. A typical static analysis process starts by representing the analysed app code as abstract models (e.g., Call Graph (CG), Control-Flow Graph (CFG), or Unified Modelling Language (UML) class/sequence diagram) based on the purpose of analysis. The literature discusses many static analysis techniques, namely:

- Control-Flow Analysis [3–5].
- Data-Flow Analysis [4,6].
- Points-to analysis [4].
- Call-Graph (CG) algorithms.
  - Class Hierarchy analysis (CHA) [7].
  - Rapid Type analysis (RTA) [8].
  - Variable Type Analysis (VTA) [9].
  - Andersen [10].
  - Steensguard [11].

The advantages of static analysis include: (a) All code and manifest file—a file containing meta-data about the application—are analysed. (b) It is fast relative to dynamic analysis. (c) It requires fewer hardware resources relative to dynamic analysis.

A lot of vulnerabilities can be detected using static analysis such as private data leaks, unauthorised access to protected or private resources and intent injection. It is also used to detect permission misuse, energy consumption, clone detection and test generation. Finally, it detects code verification and cryptography implementation issues as well [4].

2.1.2. Dynamic Analysis

Unlike static analysis, dynamic analysis is used to detect vulnerabilities that occur during run-time. Dynamic analysis is generally considered more complex than static analysis as it requires installing the applications as well as simulating the user input (i.e., inputs, touches and clicks). The most common dynamic analysis tests are; Fuzz Testing, Concolic Testing and Search-based Testing [12]. The differences between them include test effectiveness, test generation and vulnerabilities coverage.

Fuzz Testing is defined as "*an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program.*" The program is then monitored for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks [13,14].

Concolic testing is "*a hybrid software verification technique that merges both symbolic and concrete executions.*" Symbolic execution treats program variables as symbolic variables, whereas concrete execution performs testing on particular input paths [15,16].

Search-Based testing uses a meta-heuristic optimising search technique to automate a testing task such as simulated annealing and genetic algorithms [17].

2.2. Android Sources and Sinks

In order to understand how Android flows are analysed, sources and sinks are discussed in this section. Android Source APIs are the calls into resource methods returning non-constant values into the application code. For example, the *getDeviceId()* resource method is an Android source. It returns the International Mobile Equipment Identity (IMEI). The IMEI is considered non-constant as the method returns a different value on every mobile phone which implies that investigating the source code alone does not reveal this value. In contrast, a function that just reads a fixed constant value from a database is considered a resource method but, as per the definition, is not an Android source [18].

Android Sinks APIs are the calls into resource methods accepting at least one non-constant data value from the application code as a parameter, if and only if a new value is written or an existing one is overwritten on the resource. The *sendTextMessage()* resource method is an example of an Android sink as both the message text and the phone number it receives are possibly non-constants [18].

3. Related Work

In this section, the pre-eminent available vulnerability detection tools are summarised. These tools perform static or dynamic analysis or both. The summary performed below is concerned with categorising the features of the tools, deciding whether it is static or dynamic. Table 1 lists the features of those tools. Moreover, the features covered in this work, discussed later in Section 4, are marked in the last column of the table. The implemented features are discussed in details later in the paper.

Similar to AndroShield, Ostorlab [19] has the same goal, which is to be a platform used by mobile developers to check for vulnerabilities in applications via static and dynamic analysis. The free version of Ostorlab does not provide any dynamic analysis features. Although providing APIs and documentation on how to use Ostorlab platform, it lacks providing the source code, modules or architecture. Looking at the tools at Table 1, QARK [20], for example, is a static analysis tool. It is capable of finding common security vulnerabilities in Android applications. QARK detects vulnerabilities like: Improperly Protected Exported Components detection, Enable Backups Feature detection, Debuggable App Mode detection, and Exported Activities and Services detection alongside the ones detected by the static analysis of Androshield. The main advantage of Androshield over QARK is that Androshield de-compiles the APK file to Jimple, while QARK decompiles it to Java bytecode. Jimple is an intermediate representation of a Java program designed to be easier to optimize than Java bytecode. It is typed, has a concrete syntax and is based on three-address code. It includes only 15 different operations, thus simplifying flow analysis. By contrast, Java bytecode includes over 200 different operations [21].

For dynamic analysis, one of the tools that compare to Androshield is Inspeckage [22] which also offers analyzing network traffic and information gathering. Information gathering includes requested permissions, app permissions, shared libraries, checking whether the app is debuggable or not, and exported components. What makes Androshield different from Inspeckage is that Inspeckage offers dynamic analysis only and requires Xposed Framework [23] to be installed as a prerequisite while Androshield offers both static and dynamic analyses without the need for prerequisite frameworks.

All vulnerabilities covered by AndroShield are written in bold in Table 1, underlining the ones covered only by AndroShield.

Table 1. Vulnerability Detection Tools Features (Specifying the ones covered in this work).

Feature	Tool(s)	Type	Covered
Reverse APK File and Flatten Classes	APKtool [24]/backsmali [25]/DroidLegacy [26]	Static	Yes
Detect Potential Leaks of Sensitive Information on Inter-Component and Intra-Component Data Flow	FlowDroid [27]/IccTA [28]/JAADAS [29]	Static	Yes
Improperly Protected Exported Components detection	QARK [20]	Static	Yes
Enable Backups Feature detection	QARK	Static	Yes
Debuggable App Mode detection	QARK	Static	Yes
Exported Activities and Services detection	QARK	Static	Yes
Information Gathering	Androguard [30]/Inspckage [22]	Static/Dynamic	Yes
Detect Native SSL Bypass	MobSF [31]	Static	
Detect Web-view SSL Bypass	MobSF	Static	
Detect Remote Web-view Debugging/Execution (Java Script)	MobSF/QARK	Static	
Detect Hard Coded Secrets	MobSF	Static	
Detect Root Detection	MobSF	Static	
Extracts the Permission Specification from Android OS	PSCout [32]	Static	
Comparing CFG Signatures to the Control Flow Graphs of Android Methods (For Detecting Malware)	CFGScanDroid [33]	Static	
Taming Reflection	DroidRA [34]	Static	
Extract "Actionable" Data from Android Malware	Madrolyzer [35]	Static	
Compute a Risk Value Based on Apps' permissions	RiskInDroid [36]	Static	
API Misuse analysis	JAADAS	Static	
Local-Denial-of-Service analysis	JAADAS/Androwarn [37]	Static	
Bound Service Vulnerability detection	QARK	Static	
Content Provider Vulnerability detection	QARK	Static	
Static Tapjacking Mitigation detection	QARK	Static	
Improper SSL x.509 Certificate detection/validation	QARK	Static	
Notifying the Use of Sticky Intents	QARK	Static	
Weak/Improper Cryptography Use detection	QARK	Static	
Private Keys Embedded in the Source Code detection	QARK	Static	
Insecure Broadcast Intents detection	QARK	Static	
Insecurely Created Pending Intents detection	QARK	Static	
Outdated API Versions detection	QARK	Static	
Creation of World-Readable or World-Writeable Files detection	QARK	Static	
Fuzz Testing	AppsPlayGround [38]	Dynamic	Yes
Event Triggering	AppsPlayGround	Dynamic	Yes
Analyze Network Traffic	Inspckage	Dynamic	Yes
Insecure Network Requests (HTTP Requests) detection	-	Dynamic	Yes
Intent Crash detection	-	Dynamic	Yes
Method Hooking/Replacing	XposedBridge [39]	Dynamic	
Malware Sandboxing	cuckooDroid [40]/cuckoo sandbox [41]	Dynamic	
Track Privacy-Sensitive Information Leakage	AppsPlayGround	Dynamic	

Table 1. *Cont.*

Feature	Tool(s)	Type	Covered
Sensitive API monitoring	AppsPlayGround	Dynamic	
Intelligent UI execution	AppsPlayGround	Dynamic	
Prevent Sandbox detection	AppsPlayGround/cuckooDroid	Dynamic	
Kernel Level monitoring	AppsPlayGround	Dynamic	
Perform a Combination of Symoblic and Concrete Execution on the Application	ConDroid	Dynamic	

4. Methodology and Architecture

This section discusses the proposed architecture along with how static and dynamic analysis are applied to detect Android vulnerabilities.

4.1. Proposed Architecture

As shown in Figure 1, the architecture is a three-tier architecture divided into Presentation, Application and Data layers.

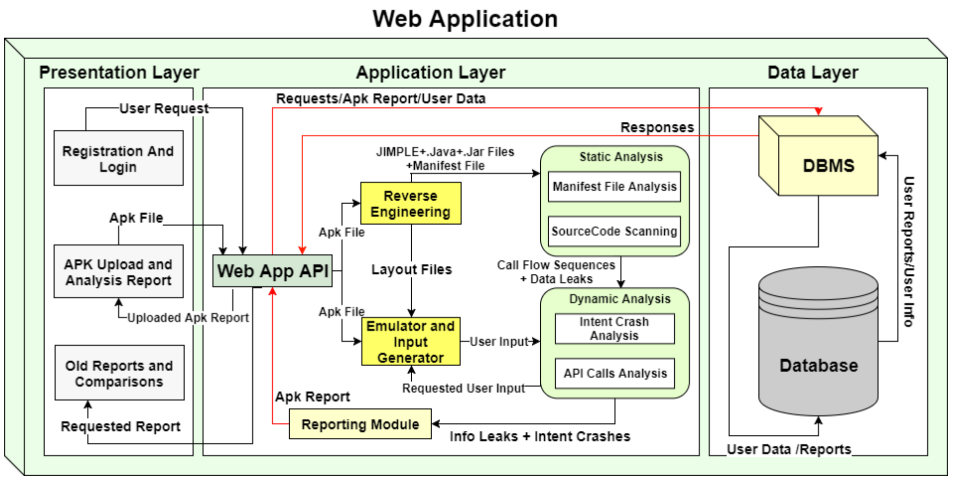


Figure 1. ANDROSHIELD Proposed Architecture.

4.1.1. Application Layer

This layer is considered the core of the business logic execution. It is divided into six modules, with each one being responsible for a single task:

(a) *Web App API* module which is responsible for handling the transfer of data between the presentation layer and the core logic.

(b) Once the user uploads the APK file, the *Reverse Engineering* module handles it. The reverse engineering process is crucial to proceed to vulnerability detection. The Jimple—simplified Java language with a fewer number of instructions—and Manifest files are extracted in this phase.

(c) Once the files are extracted, the *Static Analysis* module performs Manifest File Analysis and Source Code Scanning on the Jimple and the Manifest files to find the vulnerabilities. The found vulnerabilities (i.e., Information leaks, APK Information and permissions) are sent to the Reporting module.

(d) Before performing Dynamic Analysis, *Emulator and Input Generator* module has to simulate the user actions. The log file along with the captured APIs are sent to the Dynamic Analysis module.

(e) The *Dynamic Analysis* module performs security analysis using dynamic analysis which includes API Calls Analysis and Intent Crash Analysis. The resulting insecure internet request(s) and intent crashes are then sent to the Reporting module.

(f) Finally, the *Reporting* module compiles all the received information into a single report.

4.1.2. Presentation Layer

The Presentation layer is where the user interaction happens and it is divided into three modules that send and receive data from the application layer. The three modules are *Registration and Login*, *APK Upload and Analysis Report*, and *Old Reports* modules. The Registration and Login module is responsible for handling user permissions and account management. The APK Upload and Analysis Report module is responsible for sending the APK file to the Web App API and then display the APK report it receives. Finally, The Old Reports module handles the repository of previously generated reports by the use.

4.1.3. Data Layer

The Data layer is where user data, vulnerabilities, and APK info are stored. Old reports associated with each user are stored as well.

4.2. Methodology

- **APK Reverse Engineering**: The first step after the APK file is submitted is to de-compile (reverse engineer) it. The aim of this stage is to get the code and the manifest file of the submitted APK file. ApkAnalyzer [42]—an Android SDK command-line tool—was used to perform the reverse engineering task with the help of Soot [21]—a Java optimization framework. It is important to note here that before any application is zipped into the final APK file, the Android program (the Java source code) is compiled into a single DEX (Dalvik Executable) file. Alternatively, it will be compiled into multiple DEX files if the app and the libraries it references exceed 64KB. Then, the DEX file(s) are reversed and then transformed into JAR files—which are compiled Java files. ApkAnalyzer handles this process, it extracts the DEX files from the APK file, then transforms them into Jimple files directly by soot framework that can be used for performing the analysis. The output of this phase includes the code as well as the extracted binary manifest file.
- **Decoding Manifest File**: The output manifest file is in binary format. At this step, the manifest file is transformed from binary format to XML format.
- **Extracting Meta-data**: Having the XML version of the manifest file, the following information is extracted:

  1. Package Name
  2. Version Name
  3. Version Code
  4. Minimum SDK Version
  5. Target SDK Version
  6. Debug Mode (Yes/No)
  7. Backup Mode (Yes/No)
  8. Test Only Flag (Yes/No)

  They are then displayed in the report and some are used as well for Static Analysis phase.
- **Performing Static Analysis**: In this phase, the static analysis uses the reverse engineered code to trace vulnerabilities. During static analysis, the code is analyzed without executing the application. Details of the static analysis are discussed in Section 4.2.1.
- **Performing Dynamic Analysis**: Dynamic analysis focuses on tracing vulnerabilities while the application is running. Vulnerabilities, in this case, are not detected by analyzing the code in the static phase. Details of the dynamic analysis are discussed in Section 4.2.2.
- **Generating the Report**: The final phase is to combine all the information gathered and visualize them in a user-friendly report format.

4.2.1. Static Analysis

Static Analysis is performed by passing through two stages: (a) Manifest File Analysis and (b) Source Code Scanning.

Manifest File Analysis

This phase aims to match vulnerabilities' attribute names. The following list shows the top attributes and when to consider each one vulnerable.

- **Backup Flag** is considered vulnerable if its value is **true**. Backup is done by copying the application data (for example, application database and files) to the SD card. Application information on the SD card is less secure than on the mobile phone memory. This is because it is easier to grant write permission on SD cards. Vulnerability risk of enabling backup is estimated **Low** as the information that is backed up may not have sensitive information to be fully risky.
- **Debug Flag** is considered vulnerable if its value is **true**. Android application developers use debug mode to be able to log important fields and verify whether or not they are correct. Unfortunately, forgetting to change the application to be in release mode enables hackers to easily sniff on the application. Vulnerability risk of enabling debug mode is estimated **High** as a lot of information can be revealed while the application is in debug mode.
- **Exported Components** (activities and services) are considered vulnerable if they exist. Leaving more components opened to other applications and services requires more interfaces to secure against attacks. A malicious user can fuzz on the interfaces until finding a way to break an interface. Vulnerability risk of exported components is estimated **Low** as the getaways may be secured and may be the components are exported on purpose not by mistake, so it is not as risky as other vulnerabilities.
- **Test Only Flag** which is not vulnerable but is used for the dynamic analysis phase indicating that APK will not applicable to execute as it is just part of the application, not the whole application.

Source Code Scanning

Before discussing source code scanning to detect information leaks, sources and sinks APIs need to be specified. SuSi framework [43] provides an SVM trained model and the used model results are based on Android 4.2 version APIs. It classifies a given Android API to be a source or a sink. Working on Android 4.2 may sound deprecated but the collected APIs are not only from Android SDK libraries—which don't include runtime APIs—instead, they are included in real Android phones or Android emulators. As a result, SuSi framework provides a good base and with some hand annotation for the results fits the old and new Android APIs. SuSi provides sources and sinks of Android APIs in RIFL format—a common specification language for information-flow requirements. Next, a dummy entry point is constructed as it does not exist in Android code to be able to construct Call-Graph (CG).

This dummy entry point is constructed by FlowDroid that emulates the application life-cycle [44]. This is done such that all components (activities, services, etc.) inside an application can run in an arbitrary sequential order (including repetition). FlowDroid bases its analysis on IFDS [45], an analysis framework which is not path sensitive. Instead, it joins analysis results immediately at any control-flow merge point. It can thus generate and analyze a dummy main method in which every order of individual component life-cycles and callbacks is possible.

By scanning each source code function forward and then backwards, APIs and user-defined methods can be represented as nodes in CG and the direction of arrows in the graph shows the direction of the calls. Next comes the extraction of paths from sink APIs' nodes to source APIs' nodes. This is done by traversing the constructed CG using Depth-First Search (DFS) algorithm. Those paths from source to sink APIs are considered a tainted and a vulnerable path. They are tainted because the source API contains sensitive information about the user's Android mobile, like location, IMEI, and contacts. They are vulnerable because this sensitive information is sent outside the Android mobile by sink API like sending SMS, uploading to a remote server, and passing to another Bluetooth or NFC device. The information leak vulnerability risk is estimated **Medium** as some of the code does not indeed invoke at run time, so it may give a false positive result. Android Sources and Sinks of Android APIs are summarised in Table 2.

Throughout this work, we contributed to FlowDroid [27]—an open-source framework. It is responsible for matching the source and sinks APIs and identifying the tainted paths. However, it doesn't accept RIFL file format [46] correctly. This should be contained by Android sources and sink APIs. Therefore, we added this feature and provided an XML output file which contains the tainted paths (information leaks vulnerabilities) and its categories as shown in Table 2.

Table 2. Android APIs Sources and Sinks Categories.

Sources APIs Categories	Sinks APIs Categories
NETWORK_INFORMATION	NETWORK
LOCATION_INFORMATION	SMS
UNIQUE_IDENTIFIER	LOG
MICROPHONE	FILE
BLUETOOTH_INFORMATION	APPLICATION
FILE_INFORMATION	
CALENDAR_INFORMATION	
BROWSER_INFORMATION	
ACCOUNT_INFORMATION	
SYSTEM_SETTINGS	
APPLICATION	

4.2.2. Dynamic Analysis

There are vulnerabilities that static analysis is not able to detect. Vulnerabilities that depend on the run-time behaviour and the interaction with the application. Dealing only with reverse engineered code and code paths are not enough. To overcome this limitation, dynamic analysis is used. Dynamic analysis works on running the application and running different scenarios using randomly generated user input then monitors the running behaviour. Examples of detected behaviour are **Intent Crashing** and **Insecure Network Connections** (especially HTTP requests).

Intent Crashing could be seen as a signal for a vulnerability. The reason behind this is that it indicates (a) an open door for code injection that would run in the privileged context of the attacked application, (b) Distributed Denial of Service (DDoS) attack [47]. When malicious code is injected that is trying to access a restricted resource (i.e., camera, location, etc.), a security exception will be thrown that is not handled by the application [48] and (c) Detecting an unexpected use of an exported component [49].

Another reason for using Dynamic Analysis is that it could detect run-time anomalies regardless of the coding language being Java (native) or the developer changes to Kotlin or C++.

To be able to run an Android application, a robust Android emulator is needed, so we used and configured Android SDK emulator API 26 version (Oreo) and communicated with it through ADB [50]; a command-line utility included in the Android SDK. However, there are many architectures that the APK can support, this project supports only **x86** architecture in the phase of dynamic analysis due to performance degradation for the other architectures. After running the Android emulator, the desired APK is installed and the application is opened on the main screen (activity). If the application is labelled as "*test only*", the dynamic analysis process gets aborted. The "*test only*" flag is set by the compiler during the compilation process. This is done to speed the process while testing/running the application by compiling only a part of the application. This is considered as a limitation to the dynamic analysis process.

Next comes extracting the current views on the screen and its states i.e., clickable and non-clickable buttons. We only extract the views that the user can deal with its states in real-time like the clickable, long press and scroll-able states. Next, we loop across all the extracted states and its views and send the corresponding signal to the emulator. We used Droidbot [51] as the Android emulator and the user behaviour was generated based on the extracted states of the views. The goal of simulation is to cover the different available screens and views' states regardless of the data entered in editable text fields.

Intent Crash Analysis

While the simulation of the Android application is running on the Android emulator, information about actions, running applications, and background processes are logged. The log file from the emulator is extracted after completing the simulation and analysis by finding "**FATAL EXCEPTION**" word accompanied by the package name of the simulated application. The intent crash vulnerability risk is estimated **High** as crashing the application may enable a hacker to gain the application's permissions and cause some destruction.

API Calls Analysis

It focuses on analysing network API calls which is mainly about detecting HTTP requests. Strace [52]; a Linux API monitoring tool, is hooked to the emulator to listen to HTTP requests. The results of listening are filtered giving the URL and its source and destination hosts. The HTTP request vulnerability risk is estimated **High** as the connection is not encrypted and the user's information may be sensitive to being sniffed.

To this end, the architecture is modularized to allow it to be both extendable and adaptable to other mobile development platforms. As aforementioned, adapting to other platforms would require adjusting the reverse engineering and choosing the suitable emulator. All the required modifications are limited to the Application Layer (Section 4.1.1).

5. Experiments and Results

As mentioned above, this work performs both static and dynamic analysis on an uploaded APK. Table 3 summarises the vulnerabilities and their risk levels. Each vulnerability is ranked based on two metrics; (a) the associated security risk and (b) the impact on the user. Then, an overall risk is calculated based on these two values. For example, the "Insecure Network Request" has a *high* security risk and a *medium* impact on the user with an overall *high* risk, compared to setting the "Back-up" flag which has a *low* impact on the user and a *medium* risk resulting in a *low* overall risk.

Table 3. Vulnerability Risks Levels.

Vulnerability	Vulnerability Security Risk	Impact on User/App	Overall
Insecure Network Request	High	Medium	High
Information Leak	Low	Medium	Medium
Intent Crashes	High	High	High
Debug Flag	High	Medium	High
Exported Components	Low	Low	Low
Back-up Flag	Medium	Low	Low

To evaluate the proposed model, we used application APKs from DroidBench [53] benchmark. Moreover, we developed applications with predefined vulnerabilities to evaluate the correctness of the model. After the correctness phase, real applications are evaluated as well, such as Ampere, Brevent and GoIV Calculator.

The model is evaluated against 70 applications from DroidBench. These applications contain 93 information leaks. The model was able to detect 79 of the information leaks, resulting in 84% accuracy compared to DroidBench. Below are samples of the evaluated applications. It is important that Intent Crashes and Insecure Network Requests are fully covered in the proposed model, however, accuracy comparison is not possible since they are not included in other models/platforms.

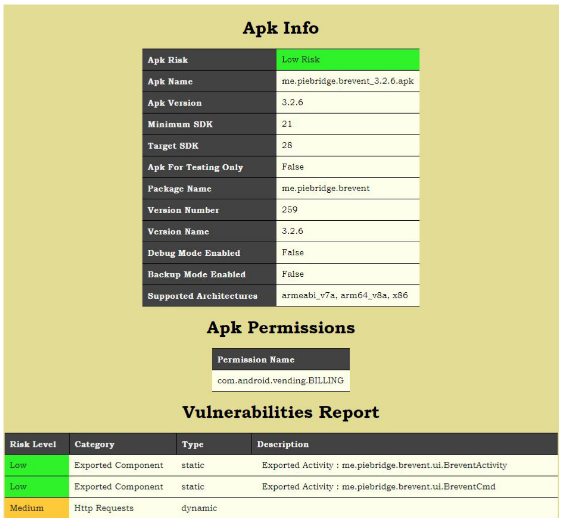
1. *httpRequest_crash application*: This is a developed application aimed to evaluate HTTP requests and intent crashes detection.
   Overall App risk: **Medium**
   The detected vulnerabilities are:

- Static: Exported Components, Debug Mode Enabled, Backup Mode Enabled
- Dynamic: Http Requests, Intent Crashes

2. *TestOnlyFlag*: This is a developed application aimed to evaluate static analysis only. Dynamic analysis won't execute because the application is flagged as **Test Only**.
Overall App risk: **Medium**
The detected vulnerabilities are:
   - Static: Exported Components, Debug Mode Enabled, Backup Mode Enabled
3. *Ampere*: This is one of the applications on the Google Play store. It is used for battery usage and voltage calculation.
Overall App risk: **Medium**
The detected vulnerabilities are:
   - Static: Exported Components, Information Leaks
4. *Brevent*: This is one of the applications on the Google Play store. It is used to control other applications that are in the background.
Overall App risk: **Low**
The detected vulnerabilities are:
   - Static: Exported Components
5. *GoIV*: This is one of the applications on the Google Play store. It is used alongside Pokemon Go game.
Overall App risk: **Low**
The detected vulnerabilities are:
   - Static: Exported Components, Information Leaks

The output of the model is an analysis report which consists of three sections. The first section is the APK information section. It contains unique information that describes the APK including the APK version, package name, minimum SDK version, target SDK version, the architectures supported by the application, whether the backup and debug modes are enabled or not, and whether the application is for testing only or not. The second section is the permissions granted to the application by the operating system. The third and final section in the report contains the vulnerabilities detected in the APK file. The vulnerabilities are represented in a table consisting of four columns: risk level, category, type, and description. The risk level indicates how severe the vulnerability is on a scale from low to high. The category indicates what the vulnerability is (e.g., exported component). The type of vulnerability specifies whether it resulted from static or dynamic analysis. Finally, the description explains the vulnerability in details.

Figure 2 shows the generated vulnerability report. Figure 2a shows a *Low Risk* report while Figure 2b shows a *High Risk* one. The three sections are highlighted. As aforementioned, the detailed description is rendered there for the use of a professional developer, however, the normal user can rely on the general information in the report which he/she can understand.



Apk Info

Apk Risk	Low Risk
Apk Name	me.piebridge.brevent_3.2.6.apk
Apk Version	3.2.6
Minimum SDK	21
Target SDK	28
Apk For Testing Only	False
Package Name	me.piebridge.brevent
Version Number	259
Version Name	3.2.6
Debug Mode Enabled	False
Backup Mode Enabled	False
Supported Architectures	armeabi_v7a, arm64_v8a, x86

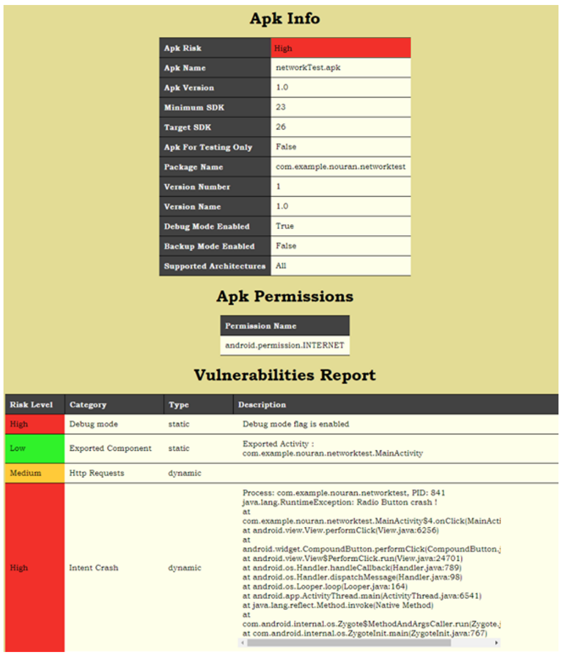
Apk Permissions

Permission Name
com.android.vending.BILLING

Vulnerabilities Report

Risk Level	Category	Type	Description
Low	Exported Component	static	Exported Activity : me.piebridge.brevent.ui.BreventActivity
Low	Exported Component	static	Exported Activity : me.piebridge.brevent.ui.BreventCmd
Medium	Http Requests	dynamic	

(**a**) Low Risk



Apk Info

Apk Risk	High
Apk Name	networkTest.apk
Apk Version	1.0
Minimum SDK	23
Target SDK	26
Apk For Testing Only	False
Package Name	com.example.nouran.networktest
Version Number	1
Version Name	1.0
Debug Mode Enabled	True
Backup Mode Enabled	False
Supported Architectures	All

Apk Permissions

Permission Name
android.permission.INTERNET

Vulnerabilities Report

Risk Level	Category	Type	Description
High	Debug mode	static	Debug mode flag is enabled
Low	Exported Component	static	Exported Activity : com.example.nouran.networktest.MainActivity
Medium	Http Requests	dynamic	
High	Intent Crash	dynamic	Process: com.example.nouran.networktest, PID: 841 java.lang.RuntimeException: Radio Button crash ! at com.example.nouran.networktest.MainActivity$4.onClick(MainActi at android.view.View.performClick(View.java:6256) at android.widget.CompoundButton.performClick(CompoundButton.j at android.view.View$PerformClick.run(View.java:24701) at android.os.Handler.handleCallback(Handler.java:789) at android.os.Handler.dispatchMessage(Handler.java:98) at android.os.Looper.loop(Looper.java:164) at android.app.ActivityThread.main(ActivityThread.java:6541) at java.lang.reflect.Method.invoke(Native Method) at com.android.internal.os.Zygote$MethodAndArgsCaller.run(Zygote.j at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:767)

(**b**) High Risk

Figure 2. Sample Vulnerabilities Report.

6. Conclusions and Future Work

The world has become very dependent on mobile applications in all aspects. With this increase in demand, the rate of releasing mobile applications has also increased to meet the continuously changing demands of the users and to keep up with competitors on the market. This has caused applications testing time to decrease and for third-party testing to become harder to perform as it requires more time and resources. The shortening of the testing phase of the released applications has caused these apps to become vulnerable to malicious users and hackers who may gain unauthorised access to private information. Throughout this paper, we proposed a usable Android vulnerability detection framework. The framework can be used by both developers and normal users. A web application is built to make it easy to use. The framework analyzes any uploaded APK file by two methods: static analysis and dynamic analysis and generates an analysis report. The types of vulnerabilities that we detected in our project were Information Leaks, Intent Crashes, Insecure Network Requests (Http Requests), Exported Android Components, Enabled Backup Mode, and Enabled Debug Mode. The framework is available on (https://github.com/AmrAshraf/AndroShield).

Future work includes improving the model by detecting more vulnerabilities. For example, detecting deprecated and vulnerable libraries used in the source code and adding support for static analysis to analyse kotlin code and not only Java code. It also involves analysing native libraries to detect malware that may cause security vulnerabilities protecting Android applications and supporting ARM architecture for APKs in the dynamic analysis since there are numerous Android mobiles and applications based on that architecture. Finally, it constitutes assessing the severity of information leaks based on linking static analysis outputs with dynamic analysis ones.

Author Contributions: Investigation, A.A., A.E., M.T.M., N.A., H.H. and I.H.; Methodology, A.A., A.E., M.T.M. and N.A.; Project administration, H.H. and I.H.; Software, A.A., A.E., M.T.M. and N.A.; Supervision, H.H. and I.H.; Visualization, A.A., A.E., M.T.M. and N.A.; Writing—original draft, A.A., A.E., M.T.M. and N.A.; Writing—review & editing, H.H. and I.H.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Clement, J. Annual number of mobile app downloads worldwide 2018 | Statista. Available online: https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/ (accessed on 12 October 2019).
2. Ponemon Institute, LLC. *The State of Mobile Application Insecurity*; Technical Report; Ponemon Institute, Sponsored by IBM: Great Lakes, MI, USA, 2015.
3. Wang, X.; Zhu, S.; Zhou, D.; Yang, Y. Droid-AntiRM: Taming Control Flow Anti-analysis to Support Automated Dynamic Analysis of Android Malware. In Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, 4–8 December 2017; pp. 350–361.
4. Li, L.; Bissyandé, T.F.; Papadakis, M.; Rasthofer, S.; Bartel, A.; Octeau, D.; Klein, J.; Traon, L. Static analysis of android apps: A systematic literature review. *Inf. Softw. Technol.* **2017**, *88*, 67–95. [CrossRef]
5. Meng, Z.; Xiong, Y.; Huang, W.; Miao, F.; Jung, T.; Huang, J. Divide and Conquer: Recovering Contextual Information of Behaviors in Android Apps around Limited-quantity Audit Logs. *arXiv* **2018**, arXiv:1809.07036.
6. Wei, F.; Roy, S.; Ou, X. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. *ACM Trans. Priv. Secur. (TOPS)* **2018**, *21*, 14. [CrossRef]
7. Dean, J.; Grove, D.; Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In Proceedings of the European Conference on Object-Oriented Programming, Åarhus, Denmark, 7–11 August 1995; pp. 77–101.
8. Bacon, D.F.; Sweeney, P.F. Fast static analysis of C++ virtual function calls. *ACM Sigplan Not.* **1996**, *31*, 324–341. [CrossRef]
9. Sundaresan, V.; Hendren, L.; Razafimahefa, C.; Vallée-Rai, R.; Lam, P.; Gagnon, E.; Godin, C. *Practical Virtual Method Call Resolution for Java*; ACM: New York, NY, USA, 2000; Volume 35.
10. Andersen, L.O. Program Analysis and Specialization for the C Programming Language. Ph.D. Thesis, University of Cophenhagen, Copenhagen, Denmark, 1994.
11. Steensgaard, B. Points-to analysis in almost linear time. In Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, FL, USA, 21–24 January 1996; pp. 32–41.
12. Mohanty, H.; Mohanty, J.; Balakrishnan, A. *Trends in Software Testing*; Springer: Berlin/Heidelberg, Germany, 2017.
13. Mahmood, R.; Esfahani, N.; Kacem, T.; Mirzaei, N.; Malek, S.; Stavrou, A. A whitebox approach for automated security testing of Android applications on the cloud. In Proceedings of the 7th International Workshop on Automation of Software Test, Zurich, Switzerland, 2–3 June 2012; pp. 22–28.
14. Godefroid, P.; Kiezun, A.; Levin, M.Y. Grammar-based Whitebox Fuzzing. In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, Tucson, AZ, USA, 7–13 June; ACM: New York, NY, USA, 2008; pp. 206–215. [CrossRef]
15. Edalat, E.; Sadeghiyan, B.; Ghassemi, F. ConsiDroid: A Concolic-based Tool for Detecting SQL Injection Vulnerability in Android Apps. *arXiv* **2018**, arXiv:1811.10448.
16. Majumdar, R.; Sen, K. Hybrid Concolic Testing. In *Proceedings of the 29th International Conference on Software Engineering*; IEEE Computer Society: Washington, DC, USA, 2007; pp. 416–426. [CrossRef]
17. McMinn, P. Search-Based Software Testing: Past, Present and Future. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Berlin, Germany, 21–25 March 2011*; IEEE Computer Society: Washington, DC, USA, 2011; pp. 153–163. [CrossRef]
18. Rasthofer, S.; Arzt, S.; Bodden, E. A Machine-Learning Approach for Classifying and Categorizing Android Sources and Sinks. In Proceedings of the 2014 Network and Distributed System Security (NDSS) Symposium, NDSS 2014, San Diego, CA, USA, 23–26 February 2014.
19. Ostorlab. Ostorlab - Continuous Mobile Application Security Testing. Available online: https://www.ostorlab.co/ (accessed on 12 October 2019).
20. LinkedIn. Tool to Look for Several Security Related Android Application Vulnerabilities. Available online: https://github.com/linkedin/qark (accessed on 6 December 2018).
21. Group, S.R. Soot—A Java Optimization Framework. Available online: https://github.com/Sable/soot (accessed on 20 November 2018).

22. Acpm. Inspeckage: Android Package Inspector—Dynamic Analysis with Api Hooks, Start Unexported Activities and More. (Xposed Module). Available online: https://github.com/ac-pm/Inspeckage (accessed on 6 December 2018).
23. Tungstwenty; Rovo89. Xposed Module Repository. 2014. Available online: https://repo.xposed.info/module/de.robv.android.xposed.installer (accessed on 12 October 2019).
24. Tumbleson, C.; Wiśniewski, R. Apktool—A Tool for Reverse Engineering 3rd Party, Closed, Binary Android Apps. Available online: https://ibotpeaches.github.io/Apktool/ (accessed on 6 December 2018).
25. Gruver, B. Smali: Smali/Baksmali. Available online: https://github.com/JesusFreke/smali (accessed on 6 December 2018).
26. Deshotels, L.; Notani, V.; Lakhotia, A. DroidLegacy: Automated Familial Classification of Android Malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop, San Diego, CA, USA, 22–24 January 2014*; ACM: New York, NY, USA, 2014; pp. 3:1–3:12. [CrossRef]
27. Group, S.S.E. FlowDroid: FlowDroid Static Data Flow Tracker. Available online: https://github.com/secure-software-engineering/FlowDroid (accessed on 20 November 2018).
28. Li, L. Soot-Infoflow-Android-Iccta: Inter-Component Communication Taint Analysis for Android. Available online: https://github.com/lilicoding/soot-infoflow-android-iccta (accessed on 6 December 2018).
29. Flankerhqd. JAADAS: Joint Advanced Defect Assesment for Android Applications. Available online: https://github.com/flankerhqd/JAADAS (accessed on 6 December 2018).
30. Androguard. Reverse Engineering, Malware and Goodware Analysis of Android Applications . . . and More (ninja !). Available online: https://github.com/androguard/androguard (accessed on 6 December 2018).
31. Framework, M.S. MobSF. Available online: https://github.com/MobSF/Mobile-Security-Framework-MobSF (accessed on 6 December 2018).
32. Au, K.W.Y.; Zhou, Y.F.; Huang, Z.; Lie, D. Pscout: Analyzing the android permission specification. In Proceedings of the 2012 ACM Conference on Computer and Communications Security, Raleigh, NC, USA, 16–18 October 2012; pp. 217–228.
33. TACIXAT. CFGScanDroid: Control Flow Graph Scanning for Android. Available online: https://github.com/TACIXAT/CFGScanDroid (accessed on 6 December 2018).
34. Li, L.; Bissyandé, T.F.; Octeau, D.; Klein, J. DroidRA: Taming Reflection to Support Whole-Program Analysis of Android Apps. In Proceedings of the The 2016 International Symposium on Software Testing and Analysis (ISSTA 2016), Saarbrücken, Germany, 18–20 July 2016.
35. Maldroid. Maldrolyzer: Simple Framework to Extract "Actionable" Data from Android Malware (C&Cs, Phone Numbers etc.). Available online: https://github.com/maldroid/maldrolyzer (accessed on 6 December 2018).
36. Merlo, A.; Georgiu, G.C. RiskInDroid: Machine Learning-Based Risk Analysis on Android. In *Proceedings of the ICT Systems Security and Privacy Protection: 32nd IFIP TC 11 International Conference, SEC 2017, Rome, Italy, 29–31 May 2017*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 538–552._36. [CrossRef]
37. Debize, T. Androwarn: Yet Another Static Code Analyzer for Malicious Android Applications. Available online: https://github.com/maaaaz/androwarn (accessed on 6 December 2018).
38. Rastogi, V.; Chen, Y.; Enck, W. AppsPlayground: Automatic security analysis of smartphone applications. In Proceedings of the third ACM conference on Data and Application Security and Privacy, San Antonio, TX, USA, 18–20 February 2013; pp. 209–220.
39. XposedBridge. The Java Part of the Xposed Framework. Available online: https://github.com/rovo89/XposedBridge (accessed on 6 December 2018).
40. Idanr. CuckooDroid—Automated Android Malware Analysis with Cuckoo Sandbox. Available online: https://github.com/idanr1986/cuckoo-droid (accessed on 6 December 2018).
41. Sandbox, C. Automated Malware Analysis. Available online: https://cuckoosandbox.org/ (accessed on 6 December 2018).
42. Developers, A. Apkanalyzer. Available online: https://developer.android.com/studio/command-line/apkanalyzer (accessed on 20 November 2018).
43. Group, S.S.E. SuSi: SuSi—Our Tool to Automatically Discover Sources and Sinks in the Android Framework. Available online: https://github.com/secure-software-engineering/SuSi (accessed on 20 November 2018).

44. Arzt, S.; Rasthofer, S.; Fritz, C.; Bodden, E.; Bartel, A.; Klein, J.; Le Traon, Y.; Octeau, D.; McDaniel, P. *Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps*; ACM Sigplan Notices; ACM: New York, NY, USA, 2014; Volume 49, pp. 259–269.
45. Reps, T.; Horwitz, S.; Sagiv, M. Precise interprocedural dataflow analysis via graph reachability. In Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, CA, USA, 23–25 January 1995; pp. 49–61.
46. Bauereiß, T.; Greiner, S.; Herda, M.; Kirsten, M.; Li, X.; Mantel, H.; Mohr, M.; Perner, M.; Schneider, D.; Tasch, M. *RIFL 1.1: A Common Specification Language for Information-Flow Requirements*; Technical Report, Technical report TUD-CS-2017-0225; TU Darmstadt: Darmstadt, Germany, 2017.
47. Gu, J.; Shen, S. CVE-2017-0780: Denial-of-Service Vulnerability Can Crash Android Messages App—TrendLabs Security Intelligence Blog, Mobile Threat Response Team. 2017. Available online: https://blog.trendmicro.com/trendlabs-security-intelligence/cve-2017-0780-denial-service-vulnerability-android-messages-app/ (accessed on 9 October 2019).
48. HackingLoops. Android Exploitation Bypassing Android Permission Model. Available online: https://www.hackingloops.com/android-exploitation-bypassing-android-permission-model/ (accessed on 12 October 2019).
49. Melamed, T. Hacking Android Apps Through Exposed Components. 2017. Available online: https://www.linkedin.com/pulse/hacking-android-apps-through-exposed-components-tal-melamed/ (accessed on 10 October 2019).
50. Developers, A. Android Debug Bridge (adb). Available online: https://developer.android.com/studio/command-line/adb (accessed on 20 November 2018).
51. Project, T.H. Droidbot: A Lightweight Test Input Generator for Android. Similar to Monkey, But with More Intelligence and Cool Features! Available online: https://github.com/honeynet/droidbot (accessed on 20 November 2018).
52. Wikipedia. Strace. Available online: https://en.wikipedia.org/wiki/Strace (accessed on 20 November 2018).
53. Fritz, C.; Arzt, S.; Rasthofer, S. DroidBench: A micro-Benchmark Suite to Assess the Stability of Taint-Analysis Tools for Android. Secure Software Engineering Group at Paderborn University and Fraunhofer IEM. Available online: https://github.com/secure-software-engineering/DroidBench (accessed on 8 October 2019).