Mutation Testing to Support the Security Testing of Android Applications

Eduardo S. M. de Vasconcelos
ICMC/USP
Universidade de São Paulo
São Carlos, SP, Brazil
eduardovasconcelos@usp.br

Marcio E. Delamaro
ICMC/USP
Universidade de São Paulo
São Carlos, SP, Brazil
delamaro@icmc.usp.br

Simone R. S. Souza
ICMC/USP
Universidade de São Paulo
São Carlos, SP, Brazil
srocio@icmc.usp.br

Abstract

The Android system has seen considerable growth in its vulnerability landscape due to an extensive application catalog catering to many user needs, many of which are security sensitive. This growth leads to an ever-increasing concern about security robustness; hence, security testing Android apps has gained substantial prominence in recent years. Many security professionals and tools specialize in security testing Android applications, but the quality of testing procedures varies significantly. In this paper, we present a preliminary study exploring the use of Mutation Testing to support Android security testing. We propose novel mutation operators, implement them in code, and conduct an experiment to evaluate their resemblance to real-world vulnerabilities. We test our mutants using a well-known open-source tool named `mobsfscan`. Our results indicate the adequacy of our operators for supporting security testing. Moreover, we reveal a potential design flaw in `mobsfscan`.

CCS Concepts

• Software and its engineering → Software testing and debugging.

Keywords

Software Testing, Mutation Testing, Security Testing, Android, Mutation Operator

1 Introduction

As of December 2023, more than 2.4 million applications were available for download on Google's official Android application repository, namely Google Play [1] [52]. Since many of these applications are published by amateur developers [3], establishing application quality safeguards becomes relevant. Many applications contain defects, and these might be severe ones [14]. Given the sensitive use cases of many contemporary Android applications, quality improvement efforts focused on the security requirements of Android applications become particularly relevant.

Indeed, it is evident that, since 2016, there has been a sheer increase in the number of security vulnerabilities reported in Android in the Common Vulnerabilities and Exposures (CVE) database [40]. Likewise, security testing for Android apps has been gaining substantial prominence ever since. Nowadays, security professionals and automated tools specialize in security testing Android apps. As diverse as these are, the quality of testing procedures varies significantly. Thus, the question arises as to how to verify whether Android application security testing procedures meet acceptable levels of quality.

[1] https://play.google.com/

In recent years, there has been a movement in the scientific community to explore Mutation Testing in Android. Mutation Testing has seen considerable success in improving the quality of Android application test suites in various contexts [50]. Despite the success achieved in applying Mutation Testing in other testing domains, there is still room for advancement in testing for Android security.

This paper introduces a preliminary study that uses Mutation Testing for Android application security testing. We begin by examining the Android vulnerability landscape to identify common vulnerabilities. Based on these findings, we propose new mutation operators for Android app security. In total, 14 mutation operators were defined, with 5 each for Java and Kotlin, and 4 for XML. We developed a Mutation Testing tool named [54] to support mutant generation. `seed-vulns` implements the 14 mutation operators and can generate mutants from an Android app's codebase.

We evaluate our mutation operators using a well-known security static analysis tool named `mobsfscan` [2]. The objective is to assess whether or not the proposed mutation operators resemble real-world vulnerabilities. We confront mutation logs with vulnerability scanning reports provided by the tool. The results indicate the adequacy of proposed mutation operators for known vulnerabilities.

With this paper, we make the following contributions:

- We propose 14 security mutation operators for Android, accounting for Java, Kotlin, and XML;
- We develop a testing tool, namely `seed-vulns`, to support the proposed mutation testing for Android security. The tool is available on GitHub [54];
- We reinforce the viability of applying Mutation Testing to Android application security testing;
- We find a potential design flaw in a well-known open-source security static analysis tool, which results in systematic false negatives, and we report such finding to its maintainer for patching;
- We identify future research opportunities in Mutation Testing applied to Android application security testing.

The remainder of this paper is divided into the following sections. Section 2 provides background on the major concepts involved in the development of this study. Our mutation operators are described in Section 3. The experimental strategy we employed to answer our research question is covered in Section 4. Section 5 covers results discussion and threats to validity. Related work is presented in Section 6 and concluding remarks are presented in Section 7.

2 Background

2.1 Android Applications

Android applications, also known as apps, are the topmost component of the Android operating system architecture. They can be pre-installed by the device manufacturer, providing basic telephony, system management functionalities, and potentially other services that give the manufacturer a competitive edge. Android device owners can also install their own apps from various sources, including Google's official Android app repository, Google Play, repositories provided by the device manufacturer, or alternative repositories like open-source ones.

The most basic Android app format is an Android Package (APK). APK files contain bytecode compiled for the Android Runtime (ART) and all other runtime resources required to run the app. During installation, these are extracted and copied to system storage [18].

Among other constructs, Android apps define what is called an application manifest. It is an XML file that describes, at a high level, the structure of the app it represents. It contains an exhaustive list of application components [17], namely activities, services, broadcast receivers, and content providers; the application permissions and the hardware and software resources necessary for the app to run. One must remember that app components are coded in high-level programming languages, such as Java and Kotlin, using the Android Application Programming Interface (API).

An activity is an app component that interacts directly with the user through a Graphical User Interface (GUI) [25]. Activities must be declared in the manifest through the `activity` element, which in turn is located within the `application` element [15] of the manifest.

A service is an app component that does not have a GUI and whose execution occurs in the background, without the need for user interaction [25].

A broadcast receiver is a component through which the app may receive intents – an object representing a message exchanged between Android applications [25] –, and which may contain a description of an operation to be performed by the receiving app [19]. Unlike activities and services, broadcast receivers do not necessarily need to be described in the manifest, and may be instantiated at runtime.

A content provider is a component through which the app, using a structured interface, can provide data to other applications [25]. Indeed, content providers resemble conventional databases in their structure and usage.

Android apps actively communicate with each other and the Android system through various Inter-Process Communication (IPC) mechanisms, such as the aforementioned intents. Intents contain an operation that the sending application intends to perform and additional information relevant to its execution [25].

2.2 Android Application Vulnerabilities

To support the decision-making regarding proposing relevant Android app security mutation operators, we have surveyed three knowledge bodies of Android application vulnerabilities, namely the OWASP Mobile Top 10 Project [43], Google's Developers Common Risks [28] and the academia [1, 4, 29, 33, 34].

We consider the Google classification for vulnerabilities the most adequate from a practical point of view. OWASP's Mobile Top 10 and most academic studies provide higher-level vulnerability descriptions, taxonomies, rankings, and classifications. Developers Common Risks, on the other hand, provides in-depth descriptions of vulnerabilities from an Android application programming point of view, which is exceptionally useful for our purpose of modeling realistic Android programming errors leading to security defects which could be labeled as vulnerabilities. Therefore, we focused on Google's classification as a primary source of information regarding Android vulnerabilities to propose mutation operators.

2.3 Mutation Testing

Mutation Testing may be understood as a software testing criterion in which, taking a computer program as input, alternative versions of the program – mutants – are generated by introducing small defects – mutations – in the original program. The mutants obtained are then tested to verify whether the set of test cases written to test the original program is capable of detecting the mutations purposefully introduced in the mutants, thus measuring the quality of the test cases [27] and aiding in its improvement [9].

Mutations are performed by special operators called mutation operators. Mutation operators model errors inherent to the software development process that result in defects in the program under analysis. Therefore, mutation operators are firmly attached to the programming language; consequently, applying Mutation Testing depends on the availability of mutation operators for the programming language and specific software domain.

3 Mutation Operators for Security Vulnerabilities

This section presents the mutation operators we propose for the context of Android application security testing. The following subsections describe each proposed mutation operator, which is named according to the mutation they refer to. Each subsection presents a security vulnerability inherent to Android apps, modeled by the proposed mutation operator.

3.1 ImproperExport Mutation Operator

When defining application components in the manifest, it is possible to specify the `exported` XML property, which can assume the values *true* or *false*.

When `exported` is set to *true*, the component can be started by other applications on the system. In this case, the component is said to be exported. On the other hand, when it is set to *false*, the component cannot be started by other applications on the system.

Improperly assigning `exported="true"` may create a vulnerability since external apps accessing the exported component may lead to the unintentional exposure of sensitive information. To avoid this vulnerability, Android secure development best practices mandate that the value of property `exported` should always be defined explicitly [16].

This vulnerability leads to the definition of mutation operator **ImproperExport**, which models the improper exposure of an application component by inadvertently specifying `exported="true"` in the manifest element that defines the component in question.

Listing 1 illustrates the definition of an Android activity component in the manifest. Listing 2 shows the mutated code of such component after operator **ImproperExport** has been applied to it. Property `exported` has its value changed from *false* to *true.*

3.2 DebuggableApplication Mutation Operator

When defining an Android app using the `application` element of the manifest, it is possible to specify the `debuggable` property as *true* or *false*. This determines whether or not it is possible to attach the app to a debugger.

When the value of `debuggable` is *true*, the system allows debugging of the app. Conversely, if the value of `debuggable` is *false,* the system does not allow debugging. Thus, assigning `debuggable` to *true* means potentially allowing access to administrative elements of the app. In contrast, such elements should remain inaccessible to individuals other than app developers [6]. Attaching to a debugger may facilitate, for example, reverse engineering an Android app or even access to sensitive data that it handles.

To mitigate this vulnerability, secure development best practices mandate that the value of `debuggable` should be assigned *false* [6]. Based on the vulnerability in question, we define operator **DebuggableApplication**, which models the improper assignment of value *true* to the `debuggable` property of the `application` element in the application manifest.

Listing 3 illustrates a fictitious Android application definition in the application manifest. Such application is not debuggable. Listing 4 shows the mutated code of such application after operator **DebuggableApplication** has been applied to it. Property `debuggable` has its value changed from *false* to *true.*

3.3 ImplicitPendingIntent Mutation Operator

A pending intent can be understood as an authorization granted by an application, say app **A**, to another application, say app **B,** so that app **B** can perform a scoped operation, enjoying the same privileges as app **A**, with such privileges being granted solely for the execution of said operation.

A pending intent is represented by an instance of `PendingIntent`, whose attributes specify the intended operation. When instantiating a pending intent and delegating it to app **B,** it is up to app **A** to specify correct attributes scoping the operation delegated to app **B.** If app **A** fails to do so, it is said to have delegated an implicit pending intent to app **B**.

Once instantiated, the `PendingIntent` is handed over to app **B**. A vulnerability arises that app **B** may at this point modify the `PendingIntent` object if it happens to be implicit, hence abusing the privileges of app **A**. This may lead to improper access to resources to which app **B** should not have access.

To mitigate this vulnerability, secure development best practices recommend specifying attributes `action`, `package`, and `component` of a `PendingIntent`. Additionally, for applications designed for Android API levels 23 or higher, and lower than 30, flag `PendingIntent.FLAG_IMMUTABLE` should be applied to the instantiated object, which instructs the system not to allow its modification [23].

We derive operator **ImplicitPendingIntent** from the vulnerability described. It models the situation where the developer does not apply flag `PendingIntent.FLAG_IMMUTABLE` to a `PendingIntent` object, allowing its modification, and potentially leading to abuse.

Listing 5 illustrates the instantiation of a `PendingIntent` object using `PendingIntent.FLAG_IMMUTABLE` in the application source code. Listing 6 shows the mutated source code after operator **ImplicitPendingIntent** has been applied to the excerpt. `PendingIntent.FLAG_IMMUTABLE` is substituted by `PendingIntent.FLAG_MUTABLE`.

Listing 1: ImproperExport: original source code.

```
<activity
    android:name=".SensitiveActivity"
    android:exported="false"
    ... />
```

Listing 2: ImproperExport: mutated source code.

```
<activity
    android:name=".SensitiveActivity"
    android:exported="true"
    ... />
```

Listing 3: DebuggableApplication: original source code.

```
<application
    android:debuggable="false" ... >
    ...
</application>
```

Listing 4: DebuggableApplication: mutated source code.

```
<application
    android:debuggable="true" ... >
    ...
</application>
```

3.4 HardcodedSecret Mutation Operator

Many apps implement cryptographic functions, such as data integrity and confidentiality. However, at times, the secure storage of cryptographic keys is neglected. Apps may store keys and secrets improperly, i.e. hardcoded in the source code, in the form of strings or arrays of bytes. Under such circumstances, the compromise of cryptographic functions is almost certain, for instance, allowing adversaries to access sensitive data protected by cryptography [22].

To model this vulnerability, we propose operator **HardcodedSecret.** It models the situation in which the app developer inserts a secret in cleartext into the source code, making it detectable by code inspection or automated scanning mechanisms capable of detecting high entropies.

Listing 7 illustrates a fictitious class definition in the application source code. After operator **HardcodedSecret** is applied to the excerpt in question, a high entropy string attribute, resembling a hardcoded cryptographic secret, is introduced. Listing 8 shows the corresponding mutated code.

Listing 5: ImplicitPendingIntent: original source code.

```
PendingIntent pendingIntent =
    PendingIntent.getActivity(
        ...
        PendingIntent.FLAG_IMMUTABLE
    );
```

Listing 6: ImplicitPendingIntent: mutated source code.

```
PendingIntent pendingIntent =
    PendingIntent.getActivity(
        ...
        PendingIntent.FLAG_MUTABLE
    );
```

Listing 7: HardcodedSecret: original source code.

```
public class SomeClass {
    ...
}
```

Listing 8: HardcodedSecret: mutated source code.

```
public class SomeClass {
    public String secret =
        "MIICWwIBAAKBgQCbtF...";
    ...
}
```

Listing 9: TapjackingFullOcclusion: original source code.

```
<Button
    ...
    android:filterTouchesWhenObscured=
        "true">
</Button>
```

Listing 10: TapjackingFullOcclusion: mutated source code.

```
<Button
    ... >
</Button>
```

3.5 Tapjacking Mutation Operators

Similarly to the clickjacking vulnerability in the context of web applications, the tapjacking vulnerability in the context of mobile applications refers to the situation in which a malicious application manages to trick the user into believing that he is tapping the smartphone screen within a certain harmless context, when in fact such interaction is carried out in a malicious context.

To carry out a tapjacking attack, the malicious application overlays the graphical elements with which it wants the user to interact with seemingly harmless graphical elements. The harmless elements are strategically positioned to lead the user to tap exactly on the desired screen regions, where the hidden malicious graphical elements are located. When the user taps on a harmless element of the screen, the malicious application propagates the user's touch to the overlaid graphical element, leading to the inadvertent issuance of commands that the user never intended.

In this study, we model three types of the tapjacking vulnerability, each one by a different mutation operator [24].

TapjackingFullOcclusion Mutation Operator: the full occlusion tapjacking vulnerability occurs when the malicious application completely overlays the malicious graphical element positioned behind the seemingly harmless graphical element. To mitigate this vulnerability, secure development best practices recommend using the `filterTouchesWhenObscured="true"` attribute in the layout configuration of the sensitive graphical element in question. This attribute prevents the propagation of touches to the graphical element when it is overlapped and can be controlled either in the static XML file that defines the layout of the element or programmatically when the application instantiates the graphical interface of the element [24].

To model the discussed vulnerability, we propose operator **TapjackingFullOcclusion,** which models the situation where the developer inadvertently omitted the aforementioned configuration in the layout of the sensitive graphical element.

Listing 9 illustrates the definition of a fictitious graphical element using `filterTouchesWhenObscured="true"`. After applying operator **TapjackingFullOcclusion**, such attribute is removed, as shown in the mutated excerpt in Listing 10.

TapjackingPartialOcclusion Mutation Operator: the partial occlusion tapjacking vulnerability is similar to the full occlusion vulnerability, differing in how the malicious application overlays the malicious graphical element. In the partial occlusion type, the overlay is not complete but merely partial.

To mitigate the described vulnerability, it is up to the developer to actively check for touch events on the sensitive graphical element, via the `public boolean dispatch TouchEvent(MotionEvent event);` method of the sensitive graphical element. If `event` contains flag `MotionEvent.FLAG_WINDOW_IS_PARTIALLY_OBSCURED`, it is up to the developer to instruct the method to return *false,* indicating that the graphical element is partially overlapped, and touch events on it should not be accepted [24]. When the developer omits this check, a potential vulnerability arises. We propose operator **TapjackingPartialOcclusion** to model the situation where the developer fails to correctly check for `MotionEvent.FLAG_WINDOW_IS_PARTIALLY_OBSCURED` before accepting a touch event.

Listing 11 illustrates a fictitious implementation of the aforementioned method. After operator **TapjackingPartialOcclusion** is applied, as shown in Listing 12, the method immediately propagates the touch event in question, without any checking whatsoever.

TapjackingSetHideOverlayWindows Mutation Operator: the last kind of tapjacking we model is named overlay windows. The Android system defines a special permission called `SYSTEM_ALERT_WINDOW` that allows applications to create windows to overlay other apps' windows. In API versions 31 or higher, developers can use the `public final void setHideOverlay Windows(boolean hide);` method, with argument *true,* to prevent windows created

by other apps from appearing over their own windows [24]. Omitting it, therefore, exposes potentially sensitive windows to tapjacking. We propose operator **TapjackingSetHideOverlayWindows** to model the situation where the developer omitted the call to `.setHideOverlayWindows(true);` when instantiating an application window.

Listing 13 illustrates a call to method `setHideOverlayWindows` providing argument *true*. After applying operator **TapjackingSetHideOverlayWindows,** the method's argument is changed from *true* to *false,* as shown in Listing 14.

Listing 11: TapjackingPartialOcclusion: original source code.

```
@Override
public boolean dispatchTouchEvent(
    MotionEvent event
) {
    ...
}
```

Listing 12: TapjackingPartialOcclusion: mutated source code.

```
@Override
public boolean dispatchTouchEvent(
    MotionEvent event
) {
    return super.dispatchTouchEvent(event);
    ...
}
```

Listing 13: TapjackingSetHideOverlayWindows: original source code.

```
...
getWindow().setHideOverlayWindows(true);
...
```

Listing 14: TapjackingSetHideOverlayWindows: mutated source code.

```
...
getWindow().setHideOverlayWindows(false);
...
```

Listing 15: PlaintextHTTP: original source code.

```
<application
    ... >
</application>
```

Listing 16: PlaintextHTTP: mutated source code.

```
<application
    ...
    android:usesCleartextTraffic="true">
</application>
```

3.6 PlaintextHTTP Mutation Operator

Sending and receiving data in plaintext HTTP is considered a security vulnerability in Android. It allows adversaries positioned between the client app and its server to eavesdrop data [21].

To prevent this vulnerability, in more recent versions of the operating system, starting from API level 28, the use of HTTP in plaintext is now disabled by default. Developers can still force its use by modifying security attribute `usesCleartextTraffic` in the application manifest. `usesCleartextTraffic` may be assigned either *true* or *false,* with *false* being the default value for API levels equal to or greater than 28. If the developer chooses to change it to *false*, there is the risk of allowing data to travel inappropriately in plaintext, hence exposing it to adversaries [20]. Therefore, we propose operator **PlaintextHTTP,** which models this situation.

Listing 15 illustrates the definition of an application in the manifest. Listing 16 shows the result of applying operator **PlaintextHTTP** to this excerpt: attribute `usesCleartextTraffic` is introduced and assigned the value *true.*

3.7 `seed-vulns` Mutation Tool

To automate the application of the proposed mutation operators, we have implemented a tool named `seed-vulns` [54]. `seed-vulns` is a command-line tool implemented in Python, capable of mutating Android apps based on their source code. Apart from XML, which is familiar to the static configuration portion of all Android apps, it also supports Java and Kotlin, the main programming languages used in modern Android application development.

The basic functioning of `seed-vulns` is as follows: it expects two inputs, namely a codebase and a list of configuration parameters defining which operators it is supposed to apply. The tool processes the codebase in search of viable mutation points and outputs one mutant codebase per viable mutation point, along with mutation logs. Such logs contain the location where each mutation was applied, the original code, the mutated code, and the identifier of the corresponding mutant. The leftmost part of Figure 1 illustrates the basic functioning of `seed-vulns`. Mutants M_1 to M_N represent mutants generated, whereas mutation logs are identified by the corresponding m_1 to m_N outputs.

For the time being, the tool is only capable of applying operators, but in future versions, we plan to enhance it to include mutant analysis and evaluation so that it more robustly supports the application of the criterion to Android apps. The tool is engineered in such a way as to automatically recognize the programming language that has been used to develop its input app and to choose mutation operators accordingly. Likewise, hybrid apps (i.e., developed partly in Java and partly in Kotlin) are supported seamlessly.

The complete set of mutation operators that `seed-vulns` supports is depicted in Table 1. Note that for operator **TapjackingFullOcclusion**, there are means to achieve the same mutation using both the Android API and static XML configurations. `seed-vulns` implements both. Thus, starting from the 8 proposed mutation operators we have previously introduced, we end up with 14 mutation operators targeted specifically at Android application security: 5 for Java, 5 equivalent ones for Kotlin, and other 4 for XML.

4 Experimental Study

To evaluate how well our operators resemble real-world vulnerabilities, we have developed a Android mutation tool named

Table 1: seed-vulns mutation operators.

Operator	Java	Kotlin	XML
ImproperExport			✓
DebuggableApplication			✓
ImplicitPendingIntent	✓	✓	
HardcodedSecret	✓	✓	
TapjackingFullOcclusion	✓	✓	✓
TapjackingPartialOcclusion	✓	✓	
TapjackingSetHideOverlayWindows	✓	✓	
PlaintextHTTP			✓

`seed-vulns`, which is publicly available on GitHub [54]. We shall generate mutants for a total of 10 open-source Android apps, perform security tests against each of their mutants using a well-known Android security testing tool, namely `mobsfscan` [2] and then analyze the vulnerability reports that result from this exercise to determine whether `mobsfscan` will detect our mutations as vulnerabilities or not. With this, we hope to answer the following research question:

RQ : Do the proposed mutation operators represent real-world vulnerabilities found in Android apps?

4.1 Hypothesis Formulation

We hypothesize that applying our proposed mutation operators to subject apps and performing security tests against the mutants that result will lead to detecting mutations as security vulnerabilities, hence attesting to the adequacy of our mutation operators and their quality regarding their ability to generate mutations that resemble real-world vulnerabilities.

Therefore, we formulate our null hypothesis as follows:

H_0 : *The mutations are not detected as vulnerabilities.*

And our alternative hypothesis:

H_A : *The mutations are detected as vulnerabilities.*

Since mutation operators are independent, we shall evaluate our hypothesis concerning each mutation operator individually. While doing so, we shall look at the ratio between the number of mutations detected as vulnerabilities and the number of mutations introduced to the mutants of each subject app. For instance, suppose that applying mutation operator **O** to app **A** results in 10 mutants (10 mutations). If security testing detects 8 out of 10 such mutations as vulnerabilities, this yields a detection/mutation ratio of 8/10 (or 0.8) for app **A**, for operator **O.**

4.2 Subject Apps

In order to carry out the experiment, we chose 10 open-source Android apps, all from the OWASP MASTG (Mobile Application Security Testing Guide) vulnerable applications catalog [44]. These are purposefully vulnerable open-source Android apps developed to serve as educational resources for mobile application security professionals. Table 2 shows the programming language in which each of these apps has been coded and a brief description of each one.

As per Table 2, 6 out of 10 subject apps have been coded in Java, while 4 have been coded in Kotlin. Notwithstanding, 6 apps are generic test apps, given that they do not attempt to mimic real Android apps. They merely implement a set of vulnerabilities purposefully. A total of 3 apps attempt to mimic banking apps. Finally, 1 app attempts to mimic a shopping app.

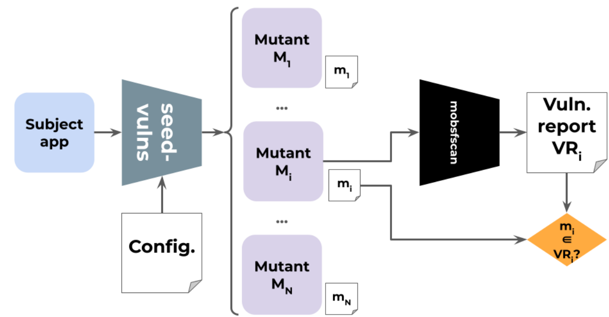

Figure 1: Evaluation execution flow.

Table 2: Subject apps list.

App	Language	Description
AndroGoat [46]	Kotlin	Generic tests
Digitalbank [48]	Java	Dummy bank
diva-android [32]	Java	Generic tests
DodoVulnerableBank [47]	Java	Dummy bank
finstergram [39]	Kotlin	Generic tests
InsecureBankv2 [49]	Java	Dummy bank
InsecureShop [7]	Kotlin	Dummy shop
MASTG-Android-Kotlin-App [42]	Kotlin	Generic tests
MASTG-Android-Java-App [42]	Java	Generic tests
ovaa [30]	Java	Generic tests

4.3 Security Testing Tool

We used `mobsfscan` [2] to search for vulnerabilities in mutants generated during the experiment. First published in 2021 as part of the renowned MobSF framework, `mobsfscan` is a static analysis tool capable of finding insecure patterns in mobile application source code.

4.4 Execution Flow

Figure 1 depicts the execution flow we used to evaluate our mutation operators. Based on input configuration parameters that never change throughout the entire experiment, we used `seed-vulns` to generate mutants of each aforementioned subject app. This process results in a set of mutant apps (from M_1 to M_N) and corresponding mutation logs (from m_1 to m_N). Each mutant M_i contains one single mutation, and each corresponding mutation log m_i stores information detailing it.

After generating mutants, we test each mutant M_i from M_1 to M_N using `mobsfscan`. A vulnerability report, namely VR_i, results from this step. VR_i details the vulnerabilities that `mobsfscan` was able to find in mutant M_i. Finally, we then analyze VR_i to determine if it contains the vulnerability introduced by m_i. We repeat this process for every subject app and their respective mutants.

5 Results and Discussion

In this section, we look at the results of generating and examining mutants. We scan each mutant for vulnerabilities using `mobsfscan`,

and we compare the resulting vulnerability reports with the mutation logs generated by `seed-vulns` during mutant generation. With this exercise, we are able to determine whether introduced mutations were detected as vulnerabilities or not.

In total, `seed-vulns` generated 286 mutants. These were created and saved, along with their respective mutation logs, for later scanning. Figure 2 shows the distribution of mutants the tool generated for each subject app, while trying to apply all 14 operators to each one. The total number of mutants generated for each app is shown in bold.

Listing 17: Original code of `.XSSActivity`.

```
<activity
    android:name=".XSSActivity"
    android:label="@string/xss" />
```

Listing 18: `.XSSActivity` mutated by *ImproperExport.*

```
<activity
    android:name=".XSSActivity"
    android:label="@string/xss"
    android:exported="true">
    <!-- Mutated by seed-vulns -->
</activity>
```

5.1 Characteristics of Mutants

By examining Figure 2, it is easy to notice that several mutation operators are not represented among generated mutants. These are **ImplicitPendingIntent** and all three tapjacking operators. Furthermore, several others are overrepresented, namely **ImproperExport** and **HardcodedSecret**.

These results are expected. Pending intents and tapjacking structures are not some of the the most frequent constructs in Android. It is natural that they do not appear in the set of subject apps, as they all have rather small codebases. Furthermore, the number of mutants generated is within the expected range, since it was not feasible to apply all operators. In addition, operators **DebuggableApplication** and **PlaintextHTTP** produce each, at most, a single mutant per application, since there exists at most one viable mutation point for them in any Android application.

As for the overrepresented operators, their high frequencies are easily explained by the fact that their mutation points are very easily found in pretty much any Android app: **ImproperExport** operates over non-exported application components, whereas **HardcodedSecret** operates over any Java or Kotlin class definition.

5.2 Detectability of Mutations as Vulnerabilities

Table 3 shows the results obtained from security scanning generated mutants using `mobsfscan`. Operators that could not be applied to any subject app, namely **ImplicitPendingIntent** and the three tapjacking operators, were omitted.

Column *Mutants* shows the total number of mutants generated for each subject app. *Detection/Mutation Ratio* shows the proportion between mutations that `mobsfscan` could detect as vulnerabilities and the total number of mutations resulting from applying each mutation operator to each subject app. To illustrate, the detection/-mutation ratio of **ImproperExport** for subject app AndroGoat is 0/23 (or 0.0), meaning that `mobsfscan` was not able to detect any out of the 23 mutants that resulted from applying operator **ImproperExport** to the subject app in question. Detection/mutation ratios of 0/0 (N/A) mean that no mutations of that type were introduced in the subject app. By inspecting Table 3, several aspects of the results stand out.

All debuggable application mutations were correctly detected as vulnerabilities. This means that we may reject the null hypothesis, and operator **DebuggableApplication** adequately resembles real-world instances of the vulnerability it models.

The same observation holds for the instances of plaintext HTTP mutations. Again, we may reject the null hypothesis. Hence, the operator **PlaintextHTTP** also resembles real-world instances of the vulnerability it models.

No instances of improper export mutations were detected as vulnerabilities. We have no means to reject the null hypothesis. Thus, technically, we must accept it and conclude that the **ImproperExport** operator is inadequate. There is a detail inherent to this operator, though, that we must bear in mind to adequately diagnose the situation. Improper exports arise from exporting *sensitive* application components, but recognizing sensitive components is not trivial. It depends on the app's context and requires human analysis. One of the subject apps, InsecureShop, is known to contain an improper export [7] (namely, exporting provider `.InsecureShopProvider`). Yet, the tool failed to detect even this occurrence. Hence, this result, along with the intrinsically semantic nature of improper exports, leads us to believe that we may not, based solely on the results herein, neither refute nor confirm the adequacy of operator **ImproperExport**. On the other hand, by examining the source code of its mutants, we may confirm that they contain the correct constructs. To illustrate, Listing 17 contains the original code of subject app AndroGoat's `.XSSActivity`, whereas Listing 18 contains the mutated code. Thus, we argue that our operator correctly models the construct that leads to the vulnerability. Nevertheless, due to the semantic nature of this vulnerability and the limitations of our tests, it was impossible to detect improper export mutations as vulnerabilities. Doing so would require different kinds of tests, perhaps involving human analysts.

By examining column HardcodedSecret of Table 3, it is evident that there were systematic inconsistencies in detecting instances of hardcoded secret mutations as vulnerabilities. In 6 out of 10 subject apps, `mobsfscan` has correctly detected all mutations as vulnerabilities, yielding a detection/mutation ratio of 1.0. In 4 out of 10 subject apps, though, the tool failed to detect all hardcoded secrets, yielding a detection/mutation ratio of 0.0. This is inconclusive. Yet, by examining the source code of generated mutants in which no mutations were detected as vulnerabilities, we notice that mutations were, indeed, applied correctly, and there is no reasonable explanation as to why the tool failed to detect them as vulnerabilities while detecting virtually the same patterns as vulnerabilities in other subject app mutants. To illustrate, Listing 19 shows the definition of subject app AndroGoat's `TrafficActivity`, in which a hardcoded secret mutation was introduced, yet remained undetected by `mobsfscan`. All instances of hardcoded secret mutations correctly detected as vulnerabilities closely resemble this one. The only reasonable explanation we could think of for this result is that there must be a design flaw in `mobsfscan`, which leads to it

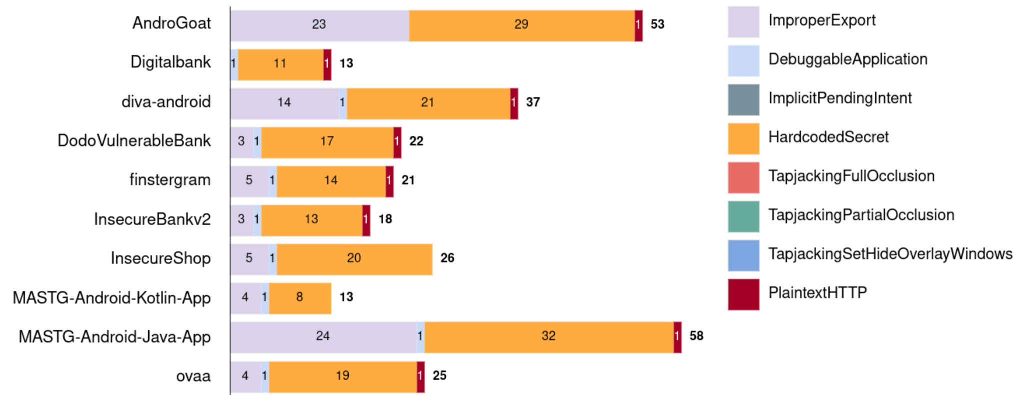

Figure 2: Mutants generated per subject app and mutation operator.

systematically failing to find hardcoded secrets in some cases. We have already contacted the maintainer of `mobsfscan` in order to report our findings and to contribute with investigating the issue and eventually patching the tool. Thus, again, we argue that our operator correctly models the construct leading to the hardcoded secret vulnerability.

Listing 19: False negative hardcoded secret.

```
class TrafficActivity :
    AppCompatActivity() {

    private val KEY =
        "a8a3abed06..." // 512 bytes
        /* Mutated by seed-vulns */
    ...
```

5.3 Threats to Validity

In order to evaluate the adequacy of our proposed operators, we have selected subject apps from a well reputable open-source vulnerable app catalog. Such apps contain fabricated vulnerabilities meant to mimic real-world ones. We assume that these are representative of real-world vulnerabilities. Yet, we cannot dismiss the possibility that we could have obtained different results have we used a different set of subject apps, perhaps one comprised of end user apps obtained from Google Play.

Another limitation of our approach is the fact that the examination of vulnerability reports and mutation logs to determine whether mutations were correctly identified as vulnerabilities was performed by a single researcher, namely the main author. Since `seed-vulns` is not yet a complete mutation testing tool, in the sense that it currently implements only mutant generation, the analysis reported herein was a semi-automated process. As much as due diligence was done in order to mitigate the risk of mistakes in the analysis and compilation of results, we cannot neglect the possibility of occasional human error.

6 Related Work

While investigating Mutation Testing for Android, we noticed that the criterion has been successfully used in testing Android apps in recent years. A systematic mapping carried out in 2020 supports this observation [51]. This mapping analyzes 16 primary studies, of which 14 have Mutation Testing as their primary focus. The first study analyzed dates back to 2015 [10], with most of the studies being conducted between 2017 and 2020 [51]. Most of the studies published in the area focus on validating/evaluating the possibility of using Mutation Testing for Android apps and proposing tools to make it viable.

One such study [10] seeks to investigate the practicality of performing Mutation Testing in Android. The authors propose 8 mutation operators, develop a prototype tool that implements them, called `muDroid`, and exercise it against an application obtained from Google Play, demonstrating the applicability of the criterion. One of the proposed operators, the APD (Activity Permission Deletion) operator, relates to application security.

In another study [13], the researchers reinforce the feasibility of adopting Mutation Testing in Android applications and propose 3 new operators specific to Android apps.

In [14], an experimental study confirms the effectiveness of applying Mutation Testing to Android. The researchers tested 9 apps using four other testing approaches in addition to Mutation Testing. They propose 5 novel mutation operators, which they then implement in the `muJava` tool [38]. The study demonstrates that Mutation Testing is more effective than other approaches.

Further studies [11, 12, 36] approach reducing the costs of Mutation Testing in Android, by discarding irrelevant operators and proposing new tools focused on test performance.

Several studies focus on developing tools for performing Mutation Testing in Android, such as `MDroid+` [35, 41]. Its authors have built a taxonomy of 262 Android defects, based on 2023 apps [35]. However, very few relate to security. From the taxonomy, 38 Java mutation operators are derived. These are implemented in `MDroid+`.

In [41], the effectiveness and performance of `MDroid+` are compared to another three Mutation Testing tools. The authors found that it outperforms the other tools in several aspects but has inferior performance regarding the total number of mutants generated. Yet, the researchers argue that the quality of its mutants is higher.

Other studies [26, 53] focus on proposing tools capable of generating mutants from compiled Android apps, hence operating directly over bytecode. The advantage of this is the possibility of applying Mutation Testing even in black box test scenarios, in which the

Table 3: Detection/Mutation ratio per mutation operator for each subject app.

App	Mutants	Detection/Mutation Ratio			
		ImproperExport	*HardcodedSecret*	*PlaintextHTTP*	*DebuggableApplication*
AndroGoat	53	0/23 (0.0)	0/29 (0.0)	1/1 (1.0)	0/0 (N/A)
Digitalbank	13	0/0 (N/A)	11/11 (1.0)	1/1 (1.0)	1/1 (1.0)
diva-android	37	0/14 (0.0)	21/21 (1.0)	1/1 (1.0)	1/1 (1.0)
DodoVulnerableBank	22	0/3 (0.0)	17/17 (1.0)	1/1 (1.0)	1/1 (1.0)
finstergram	21	0/5 (0.0)	0/14 (0.0)	1/1 (1.0)	1/1 (1.0)
InsecureBankv2	18	0/3 (0.0)	13/13 (1.0)	1/1 (1.0)	1/1 (1.0)
InsecureShop	26	0/5 (0.0)	0/20 (0.0)	0/0 (N/A)	1/1 (1.0)
MASTG-Android-Kotlin-App	13	0/4 (0.0)	0/8 (0.0)	0/0 (N/A)	1/1 (1.0)
MASTG-Android-Java-App	58	0/24 (0.0)	32/32 (1.0)	1/1 (1.0)	1/1 (1.0)
ovaa	25	0/4 (0.0)	19/19 (1.0)	1/1 (1.0)	1/1 (1.0)
	Mean	**0.0**	**0.6**	**1.0**	**1.0**
	Std.Dev	**0.0**	**0.4899**	**0.0**	**0.0**

tester does not have access to the source code. Several studies focus specifically in applying Mutation Testing to testing non-functional aspects of Android applications, such as energy consumption [31], UI [37, 45], and accessibility [50].

Other studies evaluate the quality of security static analysis tools [5, 8]. One of such studies [8] investigates whether tools contain flaws originating from poorly made design decisions which would lead them to fail to detect security vulnerabilities. The researchers propose a novel tool, called µSE, which injects security vulnerabilities into Android apps using mutation operators. The tool implements only 1 operator, modeling the situation where the app obtains sensitive information, and then "leaks" it in the system log. This type of vulnerable behavior is called data leak. The authors use µSE to exercise a static analysis tool specialized in detecting data leaks against 7 Android apps. They prove the existence of flaws in the tool. Hence, the authors demonstrate the feasibility of using Mutation Testing to verify the quality of security static analysis tools.

In another study [5], the authors increase the number of apps and static analysis tools used, totaling 15 apps and 3 tools, respectively, and implement improvements in µSE concerning the tool's ability to decide on the executability of its mutants, as well as mutant generation. The authors found new flaws in the analyzed tools.

Therefore, we were able to find a couple of mutation operators targeted at Android security (namely, the APD [10] and data leak [8] operators). Still, these appear to be collateral results of studies whose primary focus is something else. We also found a specific line of work focusing on applying Mutation Testing to analyze security static analysis tools [5, 8]. Still, so far, a single security mutation operator seems to have been developed for this endeavor. We conclude that applying Mutation Testing to Android security remains an underdeveloped field of study, and our work aims to address this research gap.

7 Concluding Remarks

In this paper, we proposed a set of mutation operators for security testing Android apps, considering the main programming languages used in this context. To support the application of the proposed mutation operators, we developed `seed-vulns`, an open-source mutation testing tool.

An experimental study was conducted to evaluate our proposed operators, and the results indicate that they can generate mutations corresponding to real-world vulnerabilities. Furthermore, we have revealed a potential design flaw in a well-known Android application security static analysis tool.

Finally, we have identified a couple of caveats and difficulties of applying Mutation Testing to the context of Android application security, namely:

- Context-dependent vulnerabilities, such as improper exports, are hard to detect based solely on automated tests, which could pose an obstacle to the further development of Mutation Testing applied to Android app security;
- Less obvious vulnerabilities, such as implicit pending intents and tapjacking, are not as widely available in open-source vulnerable Android apps. We have been unable to exercise some of our proposed operators simply because there were no viable mutation points in any of our subject apps.

In future research, we intend to address these issues by exploring the usage of Machine Learning to aid context-dependent Android vulnerability detection engines. We also plan to execute additional experiments to evaluate our mutation operators more thoroughly and explore other security static analysis tools. Also, we intend to keep incrementing our mutation operators catalog by proposing more mutation operators targeted at Android security.

Acknowledgments

Authors Marcio E. Delamaro and Simone R. S. Souza thank the support of FAPESP, process number 20/09560-2, and CNPq, under processes number 308636/2021-0 and 308445/2021-0.

References

[1] Hilmi Abdullah and Subhi R. M. Zeebaree. 2021. Android Mobile Applications Vulnerabilities and Prevention Methods: A Review. In *2021 2nd Information Technology To Enhance e-learning and Other Application (IT-ELA)*. 148–153. https://doi.org/10.1109/IT-ELA52201.2021.9773615

[2] Ajin Abraham. 2024. mobsfscan. Retrieved July 2, 2024 from https://github.com/MobSF/mobsfscan

[3] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick McDaniel, and Matthew Smith. 2016. SoK: Lessons Learned from Android Security Research for Appified Software Platforms. In *2016 IEEE Symposium on Security and Privacy (SP)*. 433–451. https://doi.org/10.1109/SP.2016.33

[4] Saket Acharya, Umashankar Rawat, Roheet Bhatnagar, and Bharat Bhushan. 2022. A Comprehensive Review of Android Security: Threats, Vulnerabilities, Malware Detection, and Analysis. *Sec. and Commun. Netw.* 2022 (jan 2022), 34 pages. https://doi.org/10.1155/2022/7775917

[5] Amit Seal Ami, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk. 2021. Systematic Mutation-Based Evaluation of the Soundness of Security-Focused Android Static Analysis Techniques. 24, 3, Article 15 (feb 2021), 37 pages. https://doi.org/10.1145/3439802

[6] AndroidDevelopers. 2024. android:debuggable. Retrieved February 5, 2024 from https://developer.android.com/privacy-and-security/risks/android-debuggable

[7] Gaurang Bhatnagar, Rujul Gandhi, and Sergey Toshin. 2022. InsecureShop. Retrieved July 2, 2024 from https://github.com/hax0rgb/InsecureShop/

[8] Richard Bonett, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk. 2018. Discovering Flaws in Security-Focused Static Analysis Tools for Android Using Systematic Mutation. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 1263–1280.

[9] M. Delamaro, J. Maldonado, and M. Jino. 2016. *Introdução Ao Teste De Software–2. ed.* Elsevier.

[10] Lin Deng, Nariman Mirzaei, Paul Ammann, and Jeff Offutt. 2015. Towards mutation analysis of Android apps. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 1–10. https://doi.org/10.1109/ICSTW.2015.7107450

[11] Lin Deng and A. Jefferson Offutt. 2018. Experimental Evaluation of Redundancy in Android Mutation Testing. *Int. J. Softw. Eng. Knowl. Eng.* 28 (2018), 1597–1618.

[12] Lin Deng and A. Jefferson Offutt. 2018. Reducing the Cost of Android Mutation Testing. In *International Conference on Software Engineering and Knowledge Engineering*.

[13] Lin Deng, Jeff Offutt, Paul Ammann, and Nariman Mirzaei. 2017. Mutation operators for testing Android apps. *Information and Software Technology* 81 (2017), 154–168. https://www.sciencedirect.com/science/article/pii/S0950584916300684

[14] Lin Deng, Jeff Offutt, and David Samudio. 2017. Is Mutation Analysis Effective at Testing Android Apps?. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 86–93. https://doi.org/10.1109/QRS.2017.19

[15] Android Developers. 2023. <activity>. Retrieved January 11, 2023 from https://developer.android.com/guide/topics/manifest/activity-element

[16] Android Developers. 2023. android:exported. Retrieved February 5, 2023 from https://developer.android.com/privacy-and-security/risks/android-exported

[17] Android Developers. 2023. App Manifest Overview. Retrieved January 11, 2023 from https://developer.android.com/guide/topics/manifest/manifest-intro

[18] Android Developers. 2023. Application Fundamentals. Retrieved January 4, 2023 from https://developer.android.com/guide/components/fundamentals

[19] Android Developers. 2023. Intent. Retrieved January 11, 2023 from https://developer.android.com/reference/android/content/Intent

[20] Android Developers. 2024. <application>. Retrieved February 5, 2024 from https://developer.android.com/guide/topics/manifest/application-element

[21] Android Developers. 2024. Cleartext / Plaintext HTTP. Retrieved February 5, 2024 from https://developer.android.com/privacy-and-security/risks/cleartext

[22] Android Developers. 2024. Hardcoded Cryptographic Secrets. Retrieved February 5, 2024 from https://developer.android.com/privacy-and-security/risks/hardcoded-cryptographic-secrets

[23] Android Developers. 2024. Pending intents. Retrieved February 5, 2024 from https://developer.android.com/privacy-and-security/risks/pending-intent

[24] Android Developers. 2024. Tapjacking. Retrieved February 5, 2024 from https://developer.android.com/privacy-and-security/risks/tapjacking

[25] J. Drake, P. Fora, Z. Lanier, C. Mulliner, S. Ridley, and G. Wicherski. 2014. *Android Hacker's Handbook.* Wiley.

[26] Camilo Escobar-Velásquez, Michael Osorio-Riaño, and Mario Linares-Vásquez. 2019. MutAPK: Source-Codeless Mutant Generation for Android Apps. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1090–1093. https://doi.org/10.1109/ASE.2019.00109

[27] R. Finkbine. 2003. Usage of mutation testing as a measure of test suite robustness. In *Digital Avionics Systems Conference, 2003. DASC '03. The 22nd* (Indianapolis, IN, USA). IEEE. https://doi.org/10.1109/DASC.2003.1245826

[28] Google. 2023. Common risks. Retrieved May 1, 2023 from https://developer.android.com/topic/security/risks

[29] Jalal B. Hur and Jawwad A. Shamsi. 2017. A survey on security issues, vulnerabilities and attacks in Android based smartphone. In *2017 International Conference on Information and Communication Technologies (ICICT)*. 40–46. https://doi.org/10.1109/ICICT.2017.8320163

[30] Oversecured Inc. 2020. ovaa. Retrieved July 2, 2024 from https://github.com/oversecured/ovaa

[31] Reyhaneh Jabbarvand and Sam Malek. 2017. µDroid: An Energy-Aware Mutation Testing Framework for Android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 208–219. https://doi.org/10.1145/3106237.3106244

[32] Aseem Jakhar. 2016. diva-android. Retrieved July 2, 2024 from https://github.com/payatu/diva-android

[33] Jignesh Joshi and Chandresh Parekh. 2016. Android smartphone vulnerabilities: A survey. In *2016 International Conference on Advances in Computing, Communication, Automation (ICACCA) (Spring)*. 1–5. https://doi.org/10.1109/ICACCA.2016.7578857

[34] X. Li, L. Yu, and X.P. Luo. 2017. Chapter 7 - On Discovering Vulnerabilities in Android Applications. In *Mobile Security and Privacy*, Man Ho Au and Kim-Kwang Raymond Choo (Eds.). Syngress, Boston, 155–166. https://www.sciencedirect.com/science/article/pii/B9780128046296000079

[35] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling Mutation Testing for Android Apps *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 233–244. https://doi.org/10.1145/3106237.3106275

[36] Jian Liu, Xusheng Xiao, Lihua Xu, Liang Dou, and Andy Podgurski. 2020. DroidMutator: An Effective Mutation Analysis Tool for Android Applications. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 77–80. https://doi.org/10.1145/3377812.3382134

[37] Eduardo Luna and Omar El Ariss. 2018. Edroid: A Mutation Tool for Android Apps. In *2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT)*. 99–108. https://doi.org/10.1109/CONISOFT.2018.8645883

[38] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability* 15, 2 (2005), 97–133. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.308 https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.308

[39] Jonas Mayer and Florentin Wieser. 2024. finstergram. Retrieved July 2, 2024 from https://github.com/netlight/finstergram

[40] MITRE. 2023. Retrieved November 18, 2022 from https://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224

[41] Kevin Moran, Michele Tufano, Carlos Bernal-Cárdenas, Mario Linares-Vásquez, Gabriele Bavota, Christopher Vendome, Massimiliano Di Penta, and Denys Poshyvanyk. 2018. MDroid+: A Mutation Testing Framework for Android. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. 33–36.

[42] OWASP. 2016. MASTG-Hacking-Playground. Retrieved July 2, 2024 from https://github.com/OWASP/MASTG-Hacking-Playground

[43] OWASP. 2023. OWASP Mobile Top 10. Retrieved March 21, 2023 from https://owasp.org/www-project-mobile-top-10/

[44] OWASP. 2024. Reference applications. Retrieved July 2, 2024 from https://mas.owasp.org/MASTG/apps/

[45] Ana C. R. Paiva, João M. E. P. Gouveia, Jean-David Elizabeth, and Márcio E. Delamaro. 2019. Testing When Mobile Apps Go to Background and Come Back to Foreground. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 102–111. https://doi.org/10.1109/ICSTW.2019.00038

[46] Satish Patnayak. 2019. AndroGoat. Retrieved July 2, 2024 from https://github.com/satishpatnayak/AndroGoat

[47] Cyber Security and Privacy Foundation (breakthesec). 2015. DodoVulnerableBank. Retrieved July 2, 2024 from https://github.com/CSPF-Founder/DodoVulnerableBank

[48] Abhinav Sejpal and Karan Sawhney. 2015. Digitalbank. Retrieved July 2, 2024 from https://github.com/CyberScions/Digitalbank

[49] Dinesh Shetty, Anant Shrivastava, and Dark Cowling. 2014. InsecureBankv2. Retrieved July 2, 2024 from https://github.com/dineshshetty/Android-InsecureBankv2

[50] Henrique Neves Silva. 2020. Uma Abordagem de Teste de Mutação para Avaliar a Acessibilidade de Aplicações Android. Retrieved July 15, 2024 from https://acervodigital.ufpr.br/handle/1884/70676 Master thesis.

[51] Henrique Neves Silva, Jackson Prado Lima, Silvia Regina Vergilio, and Andre Takeshi Endo. 2022. A mapping study on mutation testing for mobile applications. *Software Testing, Verification and Reliability* 32, 8 (2022), e1801. https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1801

[52] Statista. 2023. Number of available applications in the Google Play Store from December 2009 to December 2023. Retrieved July 3, 2024 from https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[53] Macario Polo Usaola, Gonzalo Rojas, Isyed Rodríguez, and Suilen Hernández. 2017. An Architecture for the Development of Mutation Operators. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 143–148. https://doi.org/10.1109/ICSTW.2017.31

[54] Eduardo Vasconcelos. 2024. seed-vulns. Retrieved July 2, 2024 from https://github.com/vasconcedu/seed-vulns