AppJitsu: Investigating the Resiliency of Android Applications

Onur Zungur
Boston University
Boston, USA
zungur@bu.edu

Antonio Bianchi
Purdue University
West Lafayette, USA
antoniob@purdue.edu

Gianluca Stringhini
Boston University
Boston, USA
gian@bu.edu

Manuel Egele
Boston University
Boston, USA
megele@bu.edu

***Abstract*—The Android platform gives mobile device users the opportunity to extend the capabilities of their systems by installing developer-authored apps. Companies leverage this capability to reach their customers and conduct business operations such as financial transactions. End-users can obtain custom Android applications (apps) from the Google Play, some of which are security-sensitive due to the nature of the data that they handle, such as apps from the `FINANCE` category. Although there are recommendations and standardized guidelines for secure app development with various self-defense techniques, the adoption of such methods is not mandatory and is left to the discretion of developers. Unfortunately, malicious actors can tamper with the app runtime environment and then exploit the attack vectors which arise from the tampering, such as executing foreign code with elevated privileges on the mobile platform.**

In this paper, we present AppJitsu, a dynamic app analysis framework that evaluates the resiliency of security-critical apps. We exercise the most popular 455 financial apps in attack-specific hostile environments to demonstrate the current state of resiliency against known tampering methods. Our results indicate that 25.05% of the tested apps have no resiliency against *any* common hostile methods or tools, whereas only 10.77% employed *all* defensive methods.

1. Introduction

Mobile applications (apps) are an essential part of the day-to-day activities of individuals and businesses alike. Companies develop custom apps to better reach their customer base and provide a multitude of services. For instance, in 2020 alone, 79% of smartphone owners have used their device for an online purchase [1] and conducted financial transactions.

Financial apps handle a variety of different transactions and information, many of which are sensitive, such as credit card information. It is therefore customary for app developers to put security protection mechanisms in place to thwart potential data theft threats and prevent fraud. Unfortunately, 65% of fraudulent transactions in the first quarter of 2018 were made by mobile devices, compared to 39% in 2015 [2]. Furthermore, authorities observed a recent spike in malicious actors targeting mobile banking apps [3], and around one in every 20 fraud attacks takes place thanks to a rogue mobile app [4], Recently, IBM Trusteer discovered that the problem has exacerbated to a massively scaled real-time attack campaign to steal millions of dollars from banks via mobile emulator farms [5], [6].

The variety of attack vectors and the considerably large attack surface of mobile applications (i.e., network communications, framework and app security) led to developer resources, threat recognition, and industry standards such as the Android Security Tips [7], the OWASP Top 10 Mobile Threat [8] and the OWASP Mobile App Security Testing Guide [9]. In addition, the industry provided resources for developers, such as the SafetyNet Attestation API [10], to easily integrate security solutions to their apps. However, recent studies showed a decline in the popularity of these solutions down to 11.13% in the most popular apps [11].

Among the different security solutions app developers may adopt, the OWASP guidelines suggest authors to implement self-defense mechanisms. The main purpose of this guideline is the app to detect if an app runs in a compromised environment or an attacker has tampered with the developer-authored app code. These defenses include anti-debugging mechanisms, anti-tampering protection, and root detection mechanisms.

Previous work studied the presence of these self-defense mechanisms in real-world apps. For example, Nguyen-Vu et al. [12] investigated common root detection and anti-root evasion techniques, whereas Kim et al. [13] specifically studied the resiliency of financial apps, and devised methods to bypass their self-defense mechanisms. In addition, Berlato and Mariano's [11] quantified the adoption of anti-debugging and anti-tampering protections in the most popular Android apps from the Google Play, and observed the adoption of different defensive mechanisms since 2015. However, their study used static analysis techniques, which are susceptible to errors due to obfuscation, and did not cover all the resiliency requirements set forth by the OWASP. Prior works have also extensively evaluated app hardening techniques [14], audited runtime protection mechanisms [13], or scrutinized specific defense mechanisms such as anti-root [12] or defense libraries such as ProGuard [15]. While previous studies used static analysis and focused on the usage of specific protection methods, the presence of a defense mechanism against a specific type of attack does not guarantee safety against *any* tampering attack. This is mainly due to the fact that there are multiple types of attacks available at the disposal of an attacker, each of which requires a different protection mechanism. For this reason, in contrast with previous research, we argue that there is a need to perform a comprehensive dynamic analysis study, and observe how apps behave in the presence of multiple tampering attacks.

In this paper, we study the landscape of vulnerable apps to hostile runtime environments. We are interested in answering questions such as, what percentage of security-sensitive apps employ defense mechanisms, and what is the prevalence of different resiliency capabilities. Additionally, we are interested in how and when the apps notify end-users regarding a hostile environment, and if the notifications are accurate with respect to the tampering method.

To answer such detailed questions, it is not sufficient to analyze specific app self-defense mechanisms separately. Instead, we argue that all potential runtime-attack vectors must be evaluated on a variety of different hostile environments, and we need to check if developers follow the OWASP guidelines in each of these environments.

To test the resiliency of security-sensitive apps, we build APPJITSU, a dynamic large-scale app resiliency evaluation framework. APPJITSU tests each app in different, configurable hostile runtime environments, which consist of a combination of attack vectors. Then, APPJITSU observes the behavioral differences of the app in the different tested environments, and deduces the self-defense mechanisms in place. To achieve this goal, APPJITSU uses a user-configurable combination of physical devices and emulators. In these environments, it employs different instrumentation tools to capture the app state when the it reaches a steady state. Similar to prior work which uses the information displayed on the User Interface (UI) for UI driven testing [16]–[20], APPJITSU uses the *uiautomator* tool [21] to capture the screen layout hierarchy as an indicator of the app's state. We then derive the necessary information about the implemented self-defense mechanisms after the evaluation of behavioral differences in incrementally changing hostile environments. Finally, based on APPJITSU output, we perform an analysis to detect potential defensive mechanisms implemented by the analyzed apps, or their lack thereof.

We used APPJITSU to analyze the most popular 455 apps from the FINANCE category of the Google Play [22]. Our results indicate that a striking 25.05% of the tested apps have no resiliency against *any* common hostile methods recommended in the OWASP guidelines. In contrast, only 10.77% of the apps demonstrated observable behavioral differences due to the potential threats we introduced on mobile platform runtimes. In addition to our quantitative results, we also provide a visual analysis to showcase the different behaviors that apps exhibit across various hostile environments, as well as the inaccurate messages that some of the apps display. To the best of our knowledge, our study is the first automated, dynamic-analysis-based study on how Android apps behave in different, configurable hostile environments. Furthermore, our work is the first study able to detect all resiliency requirements set forth by the OWASP guidelines. In summary, this paper makes the following contributions:

- We design and implement APPJITSU, a system that provides configurable combinations of different hostile environments to test the resiliency of apps.
- Using APPJITSU, we perform a comprehensive study to evaluate if the apps implement self-defense mechanisms.
- We analyze app behavior across different hostile environments and make the following observations: i) 25.05% of the apps have no behavioral differences against hostile platforms, contrary to 10.77% which employs all defenses, ii) 85.71% of the apps fail to detect emulated instances at least once, iii) 46.37 of apps are susceptible to repackaging attacks, while iv) 52.53%, 57.8%, 68.35% and 56.92% of the apps ran under modifiable ROM, rooted, memory hooked and debugger attached environments, respectively.

Overall, we determine that the significant majority of apps lack at least one recommended self-defense method that increase their resiliency against commonly known attack vectors. Therefore, based on our results, we recommend developers to adopt the standardized self-defense methods to thwart the commonly recognized risks against their apps.

2. Background

As a basis for the details of our proposed system APPJITSU, this section describes open standards on Android app resiliency and self-defense mechanisms, as well as a brief explanation of common tampering techniques that attackers can use.

2.1. OWASP Standards and Guides

The Open Web Application Security Project's (OWASP) Top 10 Risks [8] is a standard awareness document for developers which represents a broad consensus about the most critical security risks. Although primarily started as a list of top Web app threats, the prevalence of mobile platforms and widespread adoption of apps led to the creation of OWASP Top 10 Mobile Threats [8], which focuses on mobile apps. The most recent list from 2016 states code tampering [23] as one of the most critical risks for mobile apps. To mitigate the security threats in mobile apps, the OWASP also compiles a manual, The Mobile Security Testing Guide (MSTG) [24], which provides guidelines on how to assess the security of an app.

2.1.1. Mobile Security Testing Guide (MSTG). This comprehensive manual for mobile app security development, testing and reverse engineering provides processes, techniques and tools used by security auditors in the evaluation of a mobile app's security. Two of the most relevant sections to our paper presents i) techniques and tools for tampering and reverse engineering on Android, and ii) Android anti-reversing defenses. More specifically, anti-reversing defenses are categorized under resiliency requirements against common tampering techniques, such as rooting and hooking. For verification of testing results, these resiliency requirements are grouped under a standardized document, Mobile AppSec Verification Standard (MASVS).

2.1.2. Mobile AppSec Verification Standard (MASVS). The standards set by MASVS [9] list a series of resiliency requirements against common tampering techniques (§2.1.1). First six out of nine requirements specify the implementation of app self-defense techniques, whereas the remaining three specify how the app should react when defense mechanisms are triggered. For our work, we study the presence of defenses, and hence focus on the first six of the MSTG_RESILIENCE requirements. We show a brief summary of resiliency requirements regarding the presence of defenses along with APPJITSU designations in Table 1. All the resiliency categories, with the exception of MSTG-4, have unique requirements in defenses. As for MSTG-4, the *reverse engineering tools and frameworks* statement comprises both the root and hook tools used in MSTG-1 and MSTG-6.

2.2. Android Resiliency

Based on the OWASP recommendations and guidelines, we devise a taxonomy of the mitigations against potential security threats into 6 categories, which correspond to MSTG resiliency requirements.

2.2.1. Root Detection (MSTG-1 & 4). Rooting an Android image encompasses gaining privileged access (i.e., root) to the system. In general, a rooting framework incorporates a modified su binary, which provides access to the root user, as well as a *root manager* to provide access control to the root capabilities on the device. Two of the most common ways of root detection are to i) check the presence of the su binary in various possible locations in the file structure, or ii) to check the return value of executing su.

2.2.2. Debugger Detection (MSTG-2). The debug cycle has a critical importance during app development to analyze runtime app behavior. Development environments provide developers the means to compile apk packages with a debug flag which specifies the app to be *debuggable* (i.e., permits attaching a debugger). An attached debugger, such as ptrace-based strace or Java Debug Wire Protocol (JDWP) based tools, can arbitrarily stop app execution, inspect variables and modify memory states. Furthermore, app development environments also provide tools, such as Android Debug Bridge (adb [25]) to access and manage access to both runtime environments and the app itself. Common debugger detection techniques range from checking the return value of the isDebuggerConnected method to detecting the native process tracing utilities such as ptrace.

2.2.3. Signature Verification (MSTG-3). Signature verification ensures that the app is packaged by the developer, and hence ensures the integrity of the code base. This defense technique compares the cryptographic signature of a production release version of the app against the signature of the app on the mobile device, where discrepancies indicate package tampering.

2.2.4. Emulator Detection (MSTG-5). Android emulators allow running an app on platforms other than mobile devices via emulation of the runtime environment with Android (or derivative) system images. Emulated environments provide fast debugging, development, and modification platforms to developers and reverse-engineers alike. Emulator detection techniques can vary from emulator-specific string matching to timing-checks. Although SafetyNet Attestation API also provides methods to check the integrity of a runtime environment, similar to code integrity checks (§2.2.3), SafetyNet Attestation adoption rate decreased in recent years [11].

2.2.5. Hook Detection (MSTG-4 & 6). Hooking frameworks provide tools to execute foreign (i.e., not developer-authored) code to redirect, replace, or modify an app's control flow, which can customize the behavior of apps or provide additional functionalities. Common hook detection methods consist of identifying hooking framework-specific strings in app names or call stack traces (e.g., de.robv.android.xposed for Xposed Framework), and scanning open TCP ports for framework-operated local servers (e.g., Frida server on default port 27042).

3. Threat Model

Our threat model incorporates a benign finance-related app, which lacks one or more of the known self-defense techniques against tampering attacks. Additionally, we assume that the app runs on a hostile environment which is: i) attacker-crafted, and equipped with reverse-engineering tools, or ii) an end-user device, which has weakened security measures due to prior compromise (i.e., exploits) or user choices (e.g., rooting).

It is in the developers best interest to employ self-defending methods in security-sensitive (e.g., financial) apps to thwart tampering attacks. However, many of the self-defense mechanisms, when implemented separately, can easily be bypassed. In the first scenario, we consider malicious actors who spawn multiple instances of a finance app on hostile environments, and exploit weaknesses that arise from the lack of self-resiliency methods, to conduct their operations at scale. Such exploits ease the implementation of a large-scale campaigns for attackers, and increase the damage to protected assets, as evidenced by IBM's findings [5], [6]. In the latter scenario, the weakened security state of the user device enables potentially malicious third-party applications to access sensitive data of the finance app by the means of app-tampering methods.

We base our threat model on the comprehensive list of attack vectors that the OWASP-MSTG describes for resiliency against tampering (§ 2.2). However it is possible to augment the implementation of APPJITSU to incorporate other/additional security-requirements. Therefore, our methodology can accommodate such additions without any modifications. We present the security implications that may arise due to lack of app-resiliency in conjunction to OWASP-MSTG defined threats as follows:

Rooted Environment (MSTG-1 & 4): Root access enables code execution as the root user, and provides access to all the capabilities of the Android framework as the superuser. Effectively, an attacker or a malicious app with superuser privileges can circumvent the sandboxing feature in Android, access and alter *any* sensitive data at rest (e.g., databases or memory space) or in transmission (e.g., network communications).

Attached Debugger (MSTG-2): Similar to root capabilities, debuggers enable attackers to access app data, modify control flow, and observe app memory. This allows attackers to extract sensitive information such as credential tokens.

MSTG Category	MSTG Explanation	AppJitsu Designation
MSTG-RESILIENCE-1	The app detects, and responds to the presence of a rooted or jailbroken device either by alerting the user or terminating the app.	Anti-Root
MSTG-RESILIENCE-2	The app prevents debugging and/or detects, and responds to a debugger being attached. All available debugging protocols must be covered.	Anti-Debug
MSTG-RESILIENCE-3	The app detects, and responds to tampering with executable files and critical data within its own sandbox.	Signature Verification
MSTG-RESILIENCE-4	The app detects, and responds to the presence of widely used reverse engineering tools and frameworks on the device.	Anti-Tool (root/hook)
MSTG-RESILIENCE-5	The app detects, and responds to being run in an emulator.	Anti-Emulator
MSTG-RESILIENCE-6	The app detects, and responds to, tampering the code and data in its own memory space.	Anti-Hook

TABLE 1: MSTG Resilience requirements, explanations and APPJITSU correspondence.

Repackaged App (MSTG-3): Attackers can disassemble, modify, and repackage an `apk` package to neutralize existing app security mechanisms or craft data extraction methods within the app itself. Without the signature verification, an attacker-repackaged app would run with all modifications that compromise the app's security.

Emulated Environment (MSTG-5): An emulated runtime is an environment where the entire execution stack is under a developer or attacker's control. Attackers can defeat anti-tampering mechanisms at different levels of emulation with customized runtime environments (e.g., custom Smali emulator [26]), or massively scale their efforts [5], [6].

Hooked Functions (MSTG-4 & 6): Attackers can place hook functions to redirect security-related API calls, modify the API's return values, and hence circumvent the authentication or self-defense mechanisms on the mobile platform.

4. System Design

In this paper, we aim to subject security-sensitive apps to a variety of potentially hostile environments to determine their resilience to potential threats. As such, our main goals for the design of our framework are to: i) exercise an app to determine which hostile environments an app reacts against, ii) quantify app states and determine their relationship to the hostile runtime configurations, and iii) study the relationship between behavioral differences and self-defense techniques that an app employs (if any). Therefore, we design and implement a system with the following design criteria:

Dynamic analysis: Running an app on different runtime environments yields information on how differently an app behaves. Additionally, dynamic analysis yields information related to network activities of an app that a static analysis cannot provide.

Self-defense awareness: When an app reacts to the hostile environment, the system should be able to determine how the app reacted based on behavioral patterns.

Multiple hostile environments: Since every hostile environment can incorporate different methods to compromise an app's security, the analysis framework should provide at least one sample technique per method.

To achieve these goals, we implemented APPJITSU, a dynamic app analysis framework with multiple hostile runtime configurations to evaluate resiliency in security-sensitive apps.

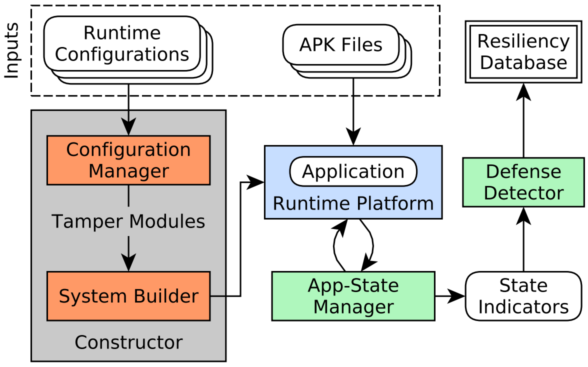


Figure 1: APPJITSU System Overview.

4.1. System Overview

Figure 1 shows the overview of APPJITSU, which comprises four main high-level components for app evaluation and data processing: i) *Configuration Manager*, ii) *System Builder*, iii) *App-State Manager*, and iv) *Defense Detector*. Within the system, the *Configuration Manager* and *System Builder* operate together to form a *Constructor* module. The *Configuration Manager* is the primary module in the *Constructor*, responsible for parsing the user-provided runtime environment configurations and selecting necessary *tamper-modules*. We define a *tamper-module* as hostile plugins, binaries, and frameworks that potential attackers can embed in their analysis environment to compromise the integrity of a system. The *System Builder* is the secondary module in the *Constructor*, which creates the runtime platform according to the specifications and *tamper-modules* that the *Configuration Manager* provides. Depending on the runtime configuration, the platform can be an emulated instance or a modification of a real hardware image. The *App-State Manager* is the runtime-platform controller and data extraction module of APPJITSU, which manages the User Interface (UI) actions during the experiment, captures and extracts the runtime-state of an app. Finally, the *Defense Detector* module receives the runtime-state of an app and compares different app-states when an app runs in different hostile environments to detect indicators of resiliency. We will now present a general overview of each module of our system, as depicted in Figure 1.

4.1.1. *Configuration Manager*. APPJITSU evaluates an app on multiple custom-built runtime environments, each having different characteristics in terms of the tools and techniques they employ. To manage and build such runtime environments, AppJitsu relies on a series of configurations and a configuration parser, which serves as the basis of *Configuration Manager* module. The *Configuration Manager* uses configuration parameters to select which tools to include in a runtime environment, and resolves dependencies or conflicts between tools and techniques. Additionally, this module ensures that configurations meet each of the AppJitsu-designated resiliency requirements from Table 1.

4.1.2. ***System Builder*.** The fundamental requirement of a dynamic analysis system is the runtime environment, and AppJitsu uses user-specified configurations to specify the characteristics of an app evaluation platform. Since the total number of configurations can be arbitrary, we use the *System Builder* module to instantiate a runtime environment according to the specific configurations from *Configuration Manager*. Additionally, *System Builder* saves the delta images[1] of each unique experimental environment to optimize storage space and avoid re-instantiation of identical runtime platforms.

4.1.3. ***App-State Manager*.** During dynamic analysis, AppJitsu needs to control an app's behavior, grant permission requests, report unresponsive apps, and finally capture the app-state indicators. One of the major indicators of an app's state is the UI elements that Android renders. Similarly, changes in the UI or pending actions, such as window slide animations or requests for user action, often indicates a state transition. Therefore, we define the *app-state indicator* as the screen layout of an app after the app reaches a steady-state (i.e., after the app completes initialization, and all permissions are granted). However, the concept of app-state can be expanded with any other relevant information for the purpose of determining system state. As AppJitsu uses app-state indicators to determine resiliency mechanisms, our system requires *App-State Manager* to control, navigate, and extract information from the UI of the runtime-platform. The *App-State Manager* module directly interacts with the runtime platform and controls the experiment to manage app installation, initialization, and further interaction.

4.1.4. ***Defense Detector*.** The main goal of AppJitsu is to study behavioral variations of app-states across different runtime platforms due to the self-defense methods that an app employs. While *App-State Manager* can extract the app-state of a particular app on a *single* runtime environment configuration, a successful behavioral comparison requires app-states from *multiple* runtime platforms. Therefore, we use the *Defense Detector* to collect multiple app-states from all runtime configurations and systematically evaluate the app-state differences.

The analysis of *Defense Detector* depends on a pairwise comparison between a baseline behavior on a default runtime environment and a deviant behavior on the hostile runtime platforms. This comparative approach provides a close approximation of the differences between a non-malicious user on a non-modified mobile platform and a malicious actor on a hostile runtime environment.

4.2. System Implementation

This section elaborates on the details of the AppJitsu prototype. Due to their inter-dependent functionalities, we chose to operate *Configuration Manager* and *System Builder* as a single *Constructor* module. We implemented *Constructor* as a combination of shell and Python scripts. In the spirit of open science and to facilitate reproducible experiments, we plan to release our implementation of AppJitsu under an open source license.

4.2.1. ***Configuration Manager*.** The *Configuration Manager* includes a custom configuration parsing module to identify the *tamper-module* requirements given a runtime-environment. An example of the configuration syntax, a sample configuration file and the respective *tamper-module* dependencies are listed in Listing 1.

```
<RUNTIME> ::= {[<FILE_INTEGRITY>] [<PLATFORM>]
     [<PRIVILEGE>] [<MEMORY_MOD>:<DEPENDENCY>]
     [<DEBUGGER>]}
<FILE_INTEGRITY> ::= (signed | repackaged)
<PLATFORM> ::= (hardware | emulator)
<PRIVILEGE> ::= (none | root:<PLATFORM>)
<MEMORY_MOD> ::= (none | frida:( <FILE_INTEGRITY> |
     <PRIVILEGE> ) )
<DEBUGGER> ::= (none | strace | jwdp)

// Example 1: baseline configuration of a hardware
     device
{[signed] [hardware] [none] [none] [none]}

// Example 2: Frida hooks on a rooted Android emulator.
{[signed] [emulator] [none] [frida:root:emulator]
     [none]}
```

Listing 1: Runtime Environment Configuration Example.

Here, we define a `<RUNTIME>` environment as a 5-tuple of the MSTG Resilience categories and their possible values within AppJitsu. Additionally, configuration parameters also specify their specific dependency modules, if any, for their integration to the runtime environment. Within AppJitsu, every tool or dependency module that *System Builder* uses to realize a runtime platform becomes part of the hostile environment, and hence called a *tamper-module*. For every configuration parameter, the *Configuration Manager* selects a *tamper-module*, which consists of binaries, frameworks, and installation scripts for the given configuration. If the *Configuration Manager* detects a dependency module which the initial configuration did not specify, it also includes *tamper-modules* of the detected dependency into the runtime environment. For instance, we use Frida [27] as a memory tampering framework (Example 2 of Listing 1), which requires root privileges[2]. Consequently, here, the Frida configuration states root access as a dependency module. Although the initial configuration did not require a rooted runtime environment (i.e., `PRIVILEGE` configuration is set to ”none”), the *Configuration Manager* still includes the necessary *tamper-module*, which enables root privileges in

[1] Modifications to an Android image result in *base* and *delta* images in the QEMU copy-on-write file format. A *delta* image only stores the changes made to the *base* image.

[2] Although it is possible to use Frida without root access, this method requires repackaging the app with `frida-gadget` shared library, and hence breaks the app integrity. Since app repackaging interferes with the `FILE_INTEGRITY` configuration of AppJitsu, we prefer a runtime platform modification to an app modification.

the runtime-environment. Finally, *Configuration Manager* collects a set of *tamper-modules* based on the environment configuration, and passes the set to the *System Builder*.

4.2.2. *System Builder*.

The *System Builder* module reads *tamper-modules* and modifies a default runtime environment in the order that the *Configuration Manager* specifies. The initial runtime-environment parameter is the `<PLATFORM>`, which, if specified as ”emulator”, requires the default runtime environment to be emulated. In this case, the *System Builder* loads an unmodified `x86` compatible Android image on the Android Emulator based on QEMU and applies necessary modifications. We use a specific version of Android image, which is capable of translating `ARM` instructions to `x86` without impacting the entire system [28]. This capability enables us to analyze `ARM` variants of the apps on our `x86` experimental platform. Otherwise, we use a Nexus 6P from Huawei as the hardware platform.

To provide privileged root access to both hardware and emulated instances of the runtime environment, we use the binaries and root manager of SuperSU v2.82. As previously mentioned, we chose Frida for the dynamic memory tampering and hooking framework due to its compatibility across different Android versions as well as the ability to be used on Android Emulator. Unfortunately, Xposed Framework [29], which is another well-known hooking framework, does not support Android versions 9.0+, and hence is incompatible with our setup. Furthermore, EdXposed [30], which is a modern replacement and variant of Xposed Framework, is also incompatible with the Android Emulator. EdXposed depends on installable modifications (i.e., modules) on top of an alternative root framework called Magisk. Magisk modules lack persistence on emulated instances, which hinders their capabilities on the Android Emulator.

Well known hooking frameworks like Xposed and Frida consist of an instrumentation layer (modified *init* process or a server on the system), and a module which specifies the memory modification target (i.e., modules which specify hooks). Our empirical analysis with a custom self-defending app showed that the presence of hooking frameworks can be detected even when there are no active method hooks present in the system.[3] Therefore, since apps can still detect the mere presence of the hooking framework, and react accordingly, the AppJitsu configuration which uses Frida instrumentation server does not have an active hook to any target.

At the end of the operations of the *Constructor* (i.e., *Configuration Manager* and *System Builder*), the AppJitsu operates a total of 6 different runtime platforms with the configurations we present in Table 2.

4.2.3. *App-State Manager*.

The *App-State Manager* module is responsible for controlling the app installation, UI interaction, and extracting the app-state indicators. During our experiments, we launch an app with Android Debug Bridge (adb) [25] `monkey` [31] with a single event injection. Similar to Bianchi et al. [32], to control the UI of both the hardware and emulators, we rely on *uiautomator* [21], which is an Android testing framework that provides a set of APIs to perform UI operations. We control the *uiautomator* through a Python wrapper [33] which connects to the device though the `adb`. During the evaluation of an app, the *App-State Manager* installs the app via `adb`, obtains UI-related information and controls the device's screen actions, such as granting permissions.

At the time of app initialization, it is possible to observe error indicators which stem from compatibility issues or runtime errors. These errors appear in system dialogs, which are separate from UI warnings that informs users of a hostile environment. Therefore, the *App-State Manager* continuously scans for such errors, and repeats the corresponding experiment if a runtime error occurs.

To determine which errors occur and what the indicators of aforementioned errors are, we conducted an initial study. Our analysis is based on the insight that an event-injection fails when there are errors in the app initialization. Therefore, we first installed apps on our hardware platform, and attempted to inject 2 events with a 10 second time delay using `adb monkey`. For all the failed event-injects, we used the *uiautomator* API to obtain the UI layout, extracted the common elements, and clustered layout hierarchies based on their text field. Then, we analyzed the common values of the clusters and extracted the indicators of errors in UI layout.

Finally, we observed three types or initialization errors: i) app not responding (ANR), which is usually an app-related crash, ii) missing Google Play components, which occurs when Google Play package is not present in the system[4], and iii) UI crashes. We present these common elements (i.e.,indicators of errors) in Listing 2.

To detect the aforementioned errors, the *App-State Manager* uses *uiautomator* API to extract and parse the UI layout in XML format, and checks for the indicators of an error. At the same time, an app which needs access to permission-protected methods requests a runtime permission request. This request displays a dialog box on the UI, which shows the resources the app is requesting access to, along with ”Allow” and ”Deny” buttons. Since Android has a centralized access management system, the dialog box stems from the Android system's package installer, and possesses a fixed layout with a deterministic resource ID within the layout hierarchy (com.android.packageinstaller). Therefore, we use the same parsing logic to detect the Allow button of a runtime permission request dialog, and grant all requested permissions until all the permissions are granted.[5] We present the resource ID of the ”Allow” button in the partial UI layout dumps in Listing 2, below the resource ID's of our error indicators.

```
<?xml version="1.0" ?> <hierarchy rotation="0"> <node

// Err1: App Not Responding
 resource-id="android:id/alertTitle"
 text="AppName has stopped"

// Err2: Missing Google Play Components
 2a: resource-id="android:id/message"
 2b: resource-id="appName:id/device_alert_info_tv"
```

[3] For Xposed Framework, hook detection with stack trace analysis still shows the framework components. An active Frida instrumentation server, on the other hand, is visible with a simple port scan. Neither of the tests had an active hook to any Java method.

[4] Google Play app is only available in a production-build Android image, which also lack some debugging capabilities.

[5] Some apps present a custom pre-permission-request dialog, which we discuss in §6.1

Runtime Platform Configuration	File Integrity	Platform	App Binary Interface	Image Build	Privilege	Memory Modification	Test Target
HW	Developer Signed	Nexus 6P	`ARM`	Production	N/A	N/A	BASELINE
HW_MOD	Re-signed	Nexus 6P	`ARM`	Production	N/A	N/A	MSTG-Resilience-3
GPLAY	Developer Signed	Emulator	`x86`	Production	N/A	N/A	MSTG-Resilience-5
GAPI	Developer Signed	Emulator	`x86`	Debug	N/A	N/A	MSTG-Resilience-2
GAPI_ROOT	Developer Signed	Emulator	`x86`	Debug	SuperSU v2.82	N/A	MSTG-Resilience-1
GAPI_FRIDA	Developer Signed	Emulator	`x86`	Debug	SuperSU v2.82	Frida	MSTG-Resilience-6
GAPI_DEBUG	Developer Signed	Emulator	`x86`	Debug	N/A	strace/JWDP	MSTG-Resilience-2

TABLE 2: APPJITSU runtime platform configurations, their properties and test targets

```
 text="AppName is missing required components and must
      be reinstalled from the Google Play."

// Err3: System UI crash
 resource-id="android:id/alertTitle" text="System UI
      isn't responding"/>

// Allow button for the permission request dialog
 resource-id="com.android.packageinstaller:id/
 permission_allow_button" text="ALLOW"

/> </node> </hierarchy>
```

Listing 2: UI Layout Element ID's in XML Format

If the *App-State Manager* detects any of the aforementioned errors on the UI, it reinstalls the app and tries to achieve a steady state (i.e., no UI errors or permission requests). Upon achieving the steady state, the *App-State Manager* captures app-state as a full set of app-state indicators. Here, a full set of app-state indicator consists of: 1) full hierarchical structure of the UI in `xml` format, and 2) a screenshot of the UI which results from rendering of the aforementioned screen layout.

Some of the apps include a screen protection mechanism, which disables screenshots of the UI when the app is on the foreground. When *App-State Manager* attempts to take a screenshot, the FrameBuffer protection mechanism produces an error which we can observe through `adb logcat`.[6] In such cases, APPJITSU uses the UI layout hierarchy in `xml` format only. Followingly, *App-State Manager* sends the set of full app-state indicators to the *Defense Detector*.

4.2.4. *Defense Detector*. The *Defense Detector* primarily acts as the detection module for app resiliency given the app-state indicators from the *App-State Manager*. More specifically, the *Defense Detector* compares the screen layout information of an app which we subject to different *tamper-modules* to observe behavioral differences. During our experiments, the hardware platform provides information on an app's *intended* state on a real device, and serves as a behavioral baseline for our framework. The remaining configurations present a hostile runtime platform with a variety of different techniques to attack a runtime-environment's integrity and trigger potential self-defense mechanisms. The *Defense Detector* compares the variations in the screenshots and captures any differences while recording which configuration caused the app-state difference. For comparison of the screenshots, we use hashes of the entire UI at the time of an app's steady state. To hash the screenshot of a steady state, we use the *Perceptional Hash* (pHash) algorithm from the *ImageHash* Python library [34], which produces the same hash for two images given that the differences between image hashes are negligibly small.[7] The reasoning behind a perceptional hash is to disregard small differences between screenshots such as the clock display. We use the default parameters for the *pHash* implementation, which produces an 8-byte locally-sensitive fuzzy hash.

The *Defense Detector* organizes the pHash of screenshots in a structured repository, which can identify any hash value given the app name and the runtime-platform configuration. If the *Defense Detector* detects differences between the hashes of the same app across all the runtime-configurations, it marks the app as an outlier and runs its detection logic. For instance, if APPJITSU captures different screenshots from non-rooted and rooted environments, *Defense Detector* evaluates the difference as root detection. The generalize, the *Defense Detector* detects the self-defense mechanism as the latest incremental change (i.e., the latest tamper-module that the system has added). Therefore, we define the parameters of defense detection logic as follows:

- P is the set of configuration parameters, where: P ∈ { integrity, platform, privilege, memory_mod, platform_access, debugger } (see Listing 1).
- C is the runtime platform configuration, where: C ∈ { P_1, P_2 ... P_α } with α = 7 for APPJITSU.
- R_C is the runtime environment with configuration C.

Given the initial configuration parameters from Table 2, APPJITSU constructs C such that C ∈ { hw, hw_{mod}, gplay, gapi, $gapi_{root}$, $gapi_{frida}$, $gapi_{debug}$ }. Finally, L_{R_C} is the amount of incremental changes to the runtime platform for configuration C, which we rank with the following inequations:

- $L_{R_{hw}} < L_{R_{hw\_mod}}$
- $L_{R_{gapi}} < L_{R_{gapi\_debug}}$
- $L_{R_{hw}} < L_{R_{gplay}} < L_{R_{gapi}} < L_{R_{gapi\_root}} < L_{R_{gapi\_frida}}$

Since APPJITSU primarily evaluates app-state indicators, we define S as the app-state indicator, where $S_{R_{C1}}$ and $S_{R_{C2}}$ denote the app-state indicators of an app on different runtime environments R_{C1} and R_{C2}. The detection logic is then defined such that:

if $(S_{R_{hw}} = S_{R_{C1}}) \wedge (S_{R_{hw}} \neq S_{R_{C2}})$ where C ∈ { hw, hw_{mod}, gplay, gapi, $gapi_{root}$, $gapi_{frida}$, $gapi_{debug}$ } ⇔ the incremental change of $L_{R_{C1}} \rightarrow L_{R_{C2}}$ is the tamper module that APPJITSU detects.

Apart from detecting the defense indicators across runtime configurations, the *Defense Detector* serves as a system error correction module for the entire APPJITSU

[6] An attempt to take a screenshot fails with the following error in adb logcat: `W/SurfaceFlinger: FB is protected: PERMISSION_DENIED`

[7] Our threshold is 20 for an 8 byte pHash output, which we empirically determined to optimize for minimum number of false positives over a corpus of 72,000 UI screenshots.

architecture. For instance, if the AppJitsu captures the screenshot in a runtime with root *and* hook, while it lacks the screenshot from root-only runtime, then the experiment on the root-only runtime may be faulty. The reasoning behind this logic is that if an app runs on an environment with higher number of modifications, it should run on environments which include only a subset of these modifications. Hence, we define the error detection logic as follows:

if $\mathrm{S}_{\mathrm{R}_{\mathrm{C1}}} = \varnothing$ while $\exists\, \mathrm{S}_{\mathrm{R}_{\mathrm{C2}}} \neq \varnothing$ for $\mathrm{L}_{\mathrm{R}_{\mathrm{C1}}} < \mathrm{L}_{\mathrm{R}_{\mathrm{C2}}}$
$\Leftrightarrow$ possible error for R_{C1}.

Defense Detector uses this error logic to notify *App-State Manager* of any potential errors in the experiment, and requests the particular experiment to be repeated. This ensures that AppJitsu has as few transient errors in app-initialization as possible.

4.3. Analysis Methodology

Our analysis mainly focuses on how many apps present different app-states for every runtime configuration of AppJitsu. With the results we obtain from this analysis, we can determine how many apps deploy self-defense methods, how prevalent different defensive behaviors in apps are, and which common defense methods are the most common. Therefore, we analyze our results with the following methodology:

First, we extract an *indicator hash set*. We define the *indicator hash set* as all the hashable app-state indicators per app (i.e., screenshot and layout hierarchy or layout hierarchy only), where every element of the set corresponds to the hash of an app-state for a given runtime environment configuration.

$$HashSet_{indicator} = [hw, hw\_mod, gplay, gapi,\\ gapi\_root, gapi\_frida, gapi\_debug]$$

We then apply a pairwise comparison between app-state hashes such that, every indicator element in the set is compared to the app-state of the hw configuration, which constitutes a baseline behavior for our purposes. If the hash difference of a compared pair is above a predetermined threshold (i.e, 20, see §4.2.4 footnotes), we mark the compared app-state with a 1, otherwise 0. At the end of our comparisons, we obtain a set of similarities (i.e., the *similarity set*) to the baseline behavior, which we encode in a binary vector. An example of a *similarity set* is shown below:

$$Set_{similarity} = [0, 1, 0, 1, 1, 1]$$

Since there are a total of 2^6 possible values[8] for a *similarity set*, we construct an $8x8$ matrix, which we call a *discrepancy matrix*, to represent all possibilities of a *similarity set*. We construct the coordinates of our *discrepancy matrix* such that, out of 6 runtime configurations, possible combinations of the first 3 represents the Y-axis, whereas the latter 3 yields the X-axis. Therefore, to determine the position of a *similarity set* in the *discrepancy matrix*, we evaluate the digits of the *similarity set* using the following formula:

$$Coordinate_y = base_{10}\left(Set_{similarity}\left[0:2\right]\right)$$
$$Coordinate_x = base_{10}\left(Set_{similarity}\left[3:5\right]\right)$$

For instance, based on our example, the similarity set above would have the coordinates $(7, 2)$.[9]

Finally, we count the number of apps that present the same *similarity sets*, and we use these values to populate the *discrepancy matrix*, which we will present in Figure 2 of §6.

5. Evaluation

We assess the ability of AppJitsu to detect app self-defense methods by testing our system on the most popular Android apps from Google Play's `FINANCE` category. We chose this category since the majority of apps handle sensitive data, such as banking credentials or account information. We also examine the inaccurate warnings for end-users which we detect during our experiments.

5.1. Experimental Setup

For app analysis on real hardware, we use Google Nexus 6P from Huawei with an ARM-compatible Android image. The emulated runtime environments run on a single computer with a octa-core Intel® Core™ i7-9750H processor, 16GB of RAM and an NVIDIA GeForce RTX 2060 Mobile GPU with 6GB GDDR6 memory. As the basis for our emulated runtime environment, we chose an `x86`-compatible Android image with ARM code translation capabilities. Our empirical analysis showed that roughly one third of the apps are not `x86` compatible (i.e., included ARM shared libraries and lacked an `x86` version), and thus we build our system to be inclusive for all the target apps.

5.2. Data Collection

5.2.1. App Collection. To collect a representative list of most popular Android apps, we used AndroidRank [35], which is a website that keeps track of the app metadata through the Google Play [22]. We then selected all the most popular apps by download count from `FINANCE` category due to their security-sensitive data handling operations. To obtain `apk` files, we used the `gplaydl` [36] package, an open-source wrapper for the reverse engineered Google Play API, and downloaded the files directly from the Google Play. For this purpose, we setup our real hardware device with a Google account, and used our account credentials as well as the specific device configuration parameters. Finally, we collected a total of 455 apps, which we all verified to run on a non-modified Android phone.

Since AppJitsu also tests the resiliency of apps against repackaging attacks, we used `apktool` [37] to disassemble `apk` files, repackage the app files and sign the repackaged app with our own cryptographic keys.

[8] Out of 7 configurations, the first value corresponds to the self-comparison of the *hw* configuration. Since this value takes a constant zero value, we ignore the first bit of information.

[9] Assuming matrix coordinates start with (0,0) on the top right corner. Note that the indices of the first 3 configurations represent the Y-axis for a more user-friendly representation.

During repackaging process, we use `apktool` with the `aapt2` [38] app packaging tool so that our repackaging process is compatible with the latest development toolsets.[10]

5.2.2. Output Data. The output of AppJitsu consists of the *app-state indicators*, which we define as the screen layout of the UI after app initialization with all the permissions granted via the built-in permission dialog. For every screen layout, we keep a database of layout hierarchy in XML format, screenshots and the perceptional hash of the screenshot. Finally, *Defense Detector* associates apps with their potential self-defense methods based on the app-state indicators.

5.3. Evaluation Strategy

We assume all the apps under scrutiny are benign apps and do not actively evade the tampering mechanisms, but rather warn the user of potential threats or weaknesses in the system. This is a reasonable assumption given that our apps are the most popular apps from the official Google Play. Prior to the experiment, we create a list of runtime environment configurations, which consists of possible combinations of all the parameters we list in Listing 1. During this process, we eliminate the combinations which create potential duplicates in the runtime environment due to the dependency requirements. Before the experiment, the AppJitsu reads the configuration files and creates an environment for every specified runtime configurations, and saves them. This ensures that we do not repeat the runtime-environment creation process for every app, and every app runs on the same set of runtime environments.

During the experiment, AppJitsu installs one app at a time for every runtime environment configuration, then executes the following actions:

1) checks for initialization errors,
2) grants permissions,
3) extracts app-state (i.e., capture screenshot and dump UI layout hierarchy in `xml` format), and
4) evaluates defense mechanisms.

If AppJitsu detects a potential error during the app initialization, it repeats the experiment process of the runtime configuration for which the error is detected. The error correction mode is a single-time process for every app, which can be triggered right after *Defense Detector* detects indicators of errors from the Listing 2 in the *App-State Manager* output. We should note that self error-correcting mechanism is useful for transient errors such as network connection problems or timing mismatches of events that *App-State Manager* injects to the app.

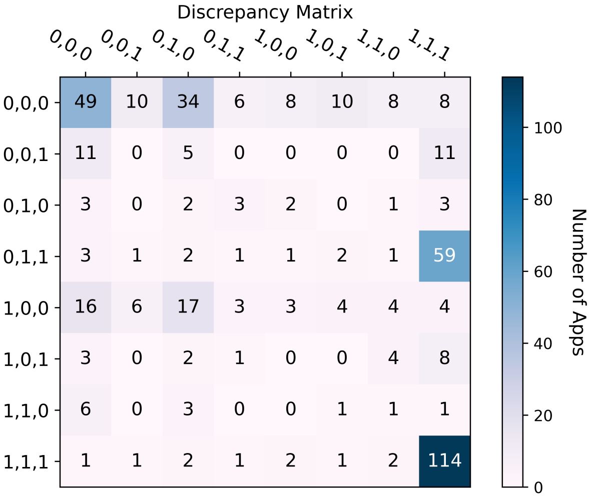
6. Results

In this section, first we use the results of our study to answer the following research questions:

- RQ1: How many apps deploy self defense mechanisms and what is the prevalence of different resiliency capabilities?

platform	hardware		emulator				
config	orig	mod	gplay	gapi	gapi _root	gapi _hook	gapi _debug
run	455	294	341	314	291	232	293
fail	0	87	114	141	164	223	162

TABLE 3: AppJitsu results in numbers.



	0,0,0	0,0,1	0,1,0	0,1,1	1,0,0	1,0,1	1,1,0	1,1,1
0,0,0	49	10	34	6	8	10	8	8
0,0,1	11	0	5	0	0	0	0	11
0,1,0	3	0	2	3	2	0	1	3
0,1,1	3	1	2	1	1	2	1	59
1,0,0	16	6	17	3	3	4	4	4
1,0,1	3	0	2	1	0	0	4	8
1,1,0	6	0	3	0	0	1	1	1
1,1,1	1	1	2	1	2	1	2	114

Figure 2: Discrepancy matrix of app *similarity sets*.

- RQ2: Which self defense methods are the most common?

Then, we present case studies about significant behaviors we observed in the analyzed apps. Table 3 presents the aggregated numerical results relative to all the device configurations tested by AppJitsu. The first column (`hw`) corresponds to a configuration in which we use a real hardware device with an unmodified app, while the second column (`hw_mod`) corresponds to a configuration in which we use a real hardware device with a repackaged app. The other four columns correspond to configurations which use emulated instances with the following device image properties.

- `gplay`: original app on stock Google Play image;
- `gapi`: modifiable image with Google APIs;
- `gapi_root`: rooted image with Google APIs;
- `gapi_hook`: rooted image with Google APIs with a running Frida instrumentation server.
- `gapi_debug`: modifiable image with Google APIs with a JDWP or `strace` attached to the app process.

AppJitsu successfully repackaged 381 apps (83.73%) in our dataset. Among these apps, 211 apps successfully launched on real hardware with the same app-state. This result shows that 46.37% of `FINANCE` apps we tested are susceptible to a repackaging attack.

To answer RQ1 and RQ2, we need a detailed breakdown of exactly how many app-state indicators have discrepancies with respect to the baseline per every combination of AppJitsu configurations. We study the prevalence of defensive behaviors based on the *discrepancy matrix* we construct with the methodology we explained in §4.3. Figure 2 presents the *discrepancy matrix* of our results.

First, we look at the top 3 most populated *similarity sets*, which are $(1, 1, 1, 1, 1, 1)$, $(0, 1, 1, 1, 1, 1)$ and $(0, 0, 0, 0, 0, 0)$. The most common case is the similarity set of $(1, 1, 1, 1, 1, 1)$, which corresponds to the case when

[10] aapt2 is enabled by default for the recent Android developer tools such as Android Studio and Android Gradle Plugin.

an app runs on *every single* AppJitsu configuration we tested. This striking result indicates that 25.05% of the most popular apps have no resiliency against *any* common hostile methods or tools. The second set $(0, 1, 1, 1, 1, 1)$ corresponds to the interesting case in which the app only detects repackaging, but lacks defensive capabilities for emulated instances. Such apps can be defeated by dynamic tampering attacks, even when the app integrity is preserved. In our dataset, this scenario occurs in 12.96% of the apps. The third most common set $(0, 0, 0, 0, 0, 0)$ corresponds to the cases in which the app *only* presents an app-state indicator iff it runs with an unmodified package *and* on real hardware. This particular behavior is an indication of high resistance against hostile environments, which we observe in 10.77% of the apps. Equivalently, 89.23% of top `FINANCE` apps do not employ at least one recommended self-defense mechanism.

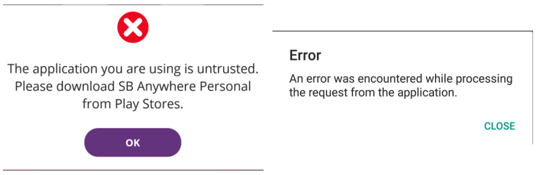
We then determine how many apps in total lack defenses against specific hostile configurations. To do so, we observe successful execution of apps on hostile environments such that the respective index of the *similarity set* indicates no observable behavioral difference. To determine the failure to detect an emulator, we select the *similarity sets* where there is at least one app-state in an emulated instance is equivalent to the baseline. We observe that 390 apps (85.71%) of the apps failed to detect emulator in at least once hostile environment that runs on Android Emulator. Using a similar technique, we identify apps which fail to detect modifiable ROM images, rooted environment, hooking framework and, finally, app process debugging. Our results indicate that 239 apps (52.53%) do not detect modifiable ROM images, 263 apps (57.8%) run despite the presence of superuser privileges, 311 apps (68.35%) fail to detect the active on-device server of the hooking process, and 259 apps (56.92%) do not detect a debugger attached to the app process.

6.1. Case Studies

In this section, we present interesting case studies found by our analysis. First, we focus on apps without any enforcement mechanisms, i.e., apps which provide no information to the user on the perils of the runtime modifications *and* still run on every hostile environment. Then, we study how different resiliency indicators influence our app-state indicators. Finally, we exemplify errors we observed in user notifications and other non-standard behaviors we found.

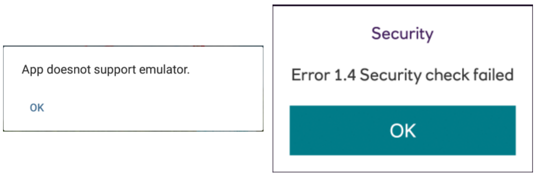
6.1.1. Defenseless Apps. In this category, we observe the lack of user notifications against *tamper-modules* of hostile environments that AppJitsu contains. We identified a set of apps that do not implement any self-defense mechanism. In fact, these apps run without issues in *every* single combination of configurations within AppJitsu. Consequently, we argue that an attacker could use any commonly known attack vector to compromise the security of these apps.

Splitwise [39], a popular finance app which enables users to record and share expenses with multiple entities and make payments via payment processors, is one example in this category. Another example is the IRS2Go [40] app. IRS2Go is the official app of the US Internal Revenue Service. This app enables users to make payments, check information related to their tax records, and generate login security codes. Due to its nature, this app handles sensitive information such as Social Security Numbers. During our experiments, we observed that both of these apps run on every runtime environment we tested, and neither have displayed any dialog box or error message to warn the users against potential threats.


The application you are using is untrusted. Please download SB Anywhere Personal from Play Stores.

Error
An error was encountered while processing the request from the application.

Figure 3: Repackaging Detection `com.sbi.SBIFreedomPlus` (left) and `com.cimbmalaysia` (right). Both apps fail to launch on the same hardware platform after repackaging and re-signing with our keys.


App doesnot support emulator.

Security
Error 1.4 Security check failed

Figure 4: Emulator Detection `com.snapwork.hdfc` (left) and `com.rbs.mobile.android.natwest` (right). Both applications display errors and fail to launch on *any* emulated instance.

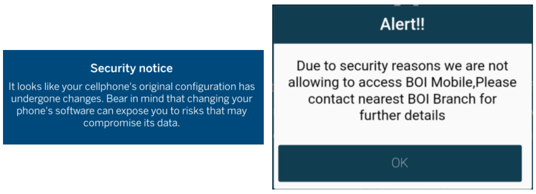
6.1.2. Signature Detection. We identify a signature detection as i) a deviation from the baseline behavior of an unmodified app on real hardware, or ii) an error during the initialization of an app.

In Figure 3, on the left we see an app showing a warning to the user, after it detects tampering of its own `apk` file. On the right, we show a case in which an app displays an error message during its initialization, after it detects being tampered. Here, AppJitsu-based tampering and the resulting modified signature caused the app's remote server to fail processing a request from the repackaged app, leading to an initialization error.

6.1.3. Emulator Detection. We detect anti-emulator behaviors by checking if i) an app refuses to run in an emulated runtime environment, or ii) an app shows a specific message to the user, complaining about being run in an emulator. In Figure 4, we show an explicit (on the left) and implicit (on the right) error message triggered by emulator detection. In both cases, the analyzed apps did not properly launch. However, in the latter case (image on the right), the error message content is non-specific, since it generically mentions the failure of a "Security check."

We will present further inconsistencies in what apps show to the user in §6.2.

6.1.4. ROM Detection. We define a ROM detection as the scenario in which an app reacts to the lack of Google Play in the operating system image, even when Google APIs are present. In Figure 5, we present how different



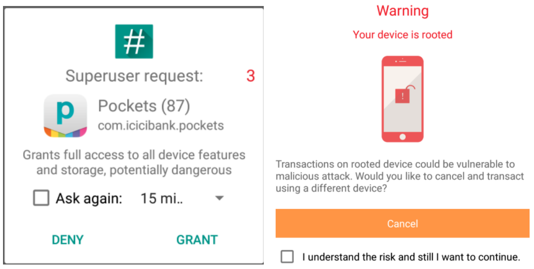
Security notice

It looks like your cellphone's original configuration has undergone changes. Bear in mind that changing your phone's software can expose you to risks that may compromise its data.

Alert!!

Due to security reasons we are not allowing to access BOI Mobile,Please contact nearest BOI Branch for further details

Figure 5: ROM Detection `com.bbva.bbvacontigo` (left) and `com.boi.mpay` (right). Both warnings appear *only* when the apps run on emulators with a modifiable Android image.



Superuser request: 3

Pockets (87)
com.icicibank.pockets

Grants full access to all device features and storage, potentially dangerous

□ Ask again: 15 mi..

Warning

Your device is rooted

Transactions on rooted device could be vulnerable to malicious attack. Would you like to cancel and transact using a different device?

□ I understand the risk and still I want to continue.

Figure 6: Root Detection. `com.icicbank.pockets` (left) attempts to execute the `su` binary, which triggers a permission request from the root manager. `com.enstage.wibmo.hdfc` (right) allows users to continue given that the user acknowledges security risks.

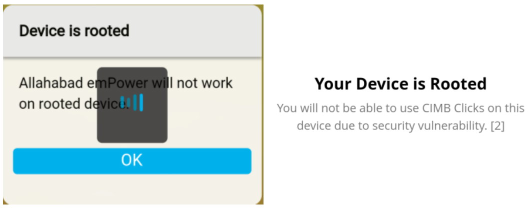
apps react to this scenario. Neither of the ROM detection errors that apps display specify the *type* of changes that the app detects. Therefore, the end-users are still oblivious to the potential threats, and uninformed if the error is because of a rooted platform or if the operating system image is merely a custom Android image without any further modifications. Here, AppJitsu's differential evaluation logic (§4.2) detects that apps display errors only when Google Play is not present on the runtime environment, and recognizes the ROM detection defense.

6.1.5. Root Detection. Our analysis shows that upon root detection, some apps warn their users, but provide them with the option to continue app execution. We show an example of this behavior in Figure 6 (right). Another root detection behavior we observed is the app's attempt to execute the `su` binary. In our testing environments, executing the `su` binary displays a pop-up window from the SuperSU app (Figure 6, left image), which is the root permission manager. As a result, AppJitsu detects this pop-up window and determines that the app which exhibits this behavior performs root detection through `su` binary execution method.

Additionally, we also found that some apps use the term *root detection* interchangeably with emulator or ROM detection, in their warning messages shown to the user. These cases will be further discussed in §6.2.

6.2. Inaccuracies in Warning Messages

During our evaluation of app-state comparisons which the *Defense Detector* marked due to the discrepancies across different runtime configurations, we discovered inaccuracies in the user-targeted warning messages. These message and notification elements conveyed the message to the user such that the app was running on a rooted runtime environment, whereas the actual platform was not. In fact, the configuration with non-rooted debug build version of Android which lacks the *tamper-modules* related SuperSU (`su` binary and the root manager app) also received the same warnings as a rooted configuration. Therefore, we see that app developers use the term *root detection* interchangeably for various resiliency methods, more prominently in emulator or ROM detection. We present two examples from two different apps in Figure 7, where all the emulated instances of these apps display a root detection warning irrespective of if the device is rooted or not.



Device is rooted

Allahabad emPower will not work on rooted device.

Your Device is Rooted

You will not be able to use CIMB Clicks on this device due to security vulnerability. [2]

Figure 7: Emulator detection with a root detection warning from `com.alb.mobilebanking` (left) and `com.cimbmalaysia` (right) apps. Both warnings appear in *all* emulated runtime platforms, regardless of the presence of root binaries on the emulator.

7. Discussion and Limitations

Our goal in this paper is to investigate the indicators of self-defense in Android applications against hostile environments. Here, we explain the corner-cases we observed during our systematic analysis along with their respective examples of the rendered UI elements.

7.1. Detection of Defenses

AppJitsu heavily relies on hashable app-state indicators in the form of screenshots, for both resiliency detection and behavioral analysis. One of the limitations of AppJitsu is that it evaluates the failure to notify users against hostile environments and successful execution alongside *tamper-modules* as lack of resiliency. The main reasoning behind this method is due to one major underlying assumption: apps in our dataset are benign and do not benefit from stealthy detection of the hostile environment. As we focus on the finance apps which handle sensitive information, the benefits of avoiding reverse engineering, tampering, and privilege escalation tools outweighs the inconvenience that an app may cause to end-users.

Unfortunately, a successful execution on AppJitsu's hostile environments may not always indicate a missing self-defense mechanism. Although such a behavior would not benefit either party in the app ecosystem, it is entirely possible that, by design, there are no indicators of detection visible to the user, or the developer, in any form, such as warning messages, failed app initialization, or app logs.

Another issue that arises from self-defense techniques is *how* a detection mechanism works. For instance, a root detection mechanisms which rely on executing code as root user may not be effective, unlike detecting root by the presence of a root manager app in the runtime environment. In the former case, AppJitsu does not automatically grant root privileges to an app, and the app would therefore fail to execute code as the root user, leading to a failed anti-root defense. However in the latter case, the app would be able to detect the presence of root-related tools,and succeed in the self-defense logic evaluation.

7.2. Method Coverage

AppJitsu cannot detect an app which uses self-defense mechanisms only after complex user interactions. While AppJitsu exercises apps even after their initialization to elicit their different functionalities, it cannot guarantee to dynamically cover all the code that an app can potentially execute. Likewise, AppJitsu cannot detect an app that performs self-defense checks but does not change its behavior in any way in response to these checks. However, we expect that most of the analyzed apps perform their self-defense checks and exhibit behavioral differences immediately after their initialization. In fact, it is more beneficial for self-defending apps to warn users against a hostile environment or deploy countermeasures as soon as possible. By doing so, an app can avoid that a user inserts sensitive information to the app which runs in a potentially-compromised environment.

Consequently, we expect apps to conform with the aforementioned principle and warn users during initialization phase. We claim that, in most of the cases, it is sufficient to observe the steady state of an app after initialization, and any further exercising of the app's functionality would not yield extra information. Hence, our results are not strongly coupled with the total amount of executed app-code, but directly tied to the code which is executed during app initialization.

7.3. False Positives and False Negatives

We evaluated a random selection of 25 apps on all hostile environments to determine false positives (PF) and false negatives (FN). To evaluate FPs, we selected the apps which AppJitsu determined to have a defense mechanism, and then manually inspected the nature of behavioral differences between baseline and hostile runtimes. We determined the cause of FPs to be UI inconsistencies, and based on our *Consistency Detector* results, we determined that AppJitsu has a 5.5% False Discovery Rate.

To evaluate FNs, we select apps which ran on AppJitsu without behavioral differences for OWASP-MSG related hostile platforms. We then manually subject these apps to the OWASP-MSTG related attack vectors and further explore the app states. Based on our evaluation, we determined that, security-related warnings can trigger due to unexplored UI states (§7.4.5 and §7.4.3), such as 1) custom permission request dialogues, 2) skipping introductory pages and, 3) login attempts. We conclude that ∼8% of the apps we tested had a FN due to unexplored UI states, where the self-defense mechanisms manifest themselves after aforementioned user interactions. We consider the cases related to the state exploration of apps to be outside the scope of our work.

7.4. Efficacy of UI-based detection

7.4.1. UI-based defenses. Prior research has demonstrated the effectiveness of UI-obfuscation against automated tools [41]. However, these obfuscation methods fall outside the scope of the current MSTG guidelines, as this defense has a narrow focus on safeguarding information on the UI only.

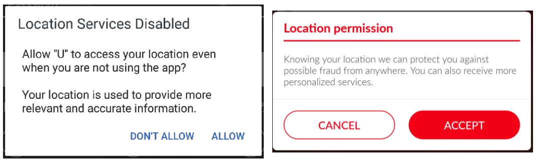
7.4.2. Dynamic Content and Non-Determinism. If an app deploys dynamically changing content (i.e., non-constant among app's different executions), AppJitsu cannot capture the consistent steady-state of the app. Dynamic, full-page advertisements (ads) are a common cause of this behavior. However, they are uncommon for official apps of financial institutions. As for third-party apps, we have only observed banner ads (i.e., a single ad bar at the bottom), which we render ineffective with our thresholding approach used by the perceptional image hashing.

To detect such cases of UI non-determinism, we implemented a *Consistency Detector* module, which runs an app on the same baseline runtime environment three times consecutively, and observes UI steady-states. The module compares perceptional hashes of displayed elements, and checks if the app consistently displays the same UI across different runs. We evaluated our dataset with the *Consistency Detector*, and observed that 4% of the apps demonstrate non-deterministic content on their steady-state due to dynamically changing content. An additional 1.5% of all apps had device configuration related inconsistencies, such as Android version, which can vary based on implementation details and cause non-deterministic UI.

7.4.3. UI of Unexplored States. As we mentioned in §7.2, AppJitsu evaluates the UI of the app after launch, and hence performs a shallow state exploration. Therefore, our evaluation is limited to defensive mechanisms that occur in the steady-state of the initialization page. However, as we demonstrated in §7.3, certain app designs allow a UI state which triggers self-defense mechanisms after users take a certain action (e.g., click on ”Login” button). As the app's UI states can be arbitrarily complex, a defensive mechanism that triggers on a UI state other than the initial state would avoid AppJitsu's detection (i.e., delayed response). We consider state exploration of app states to fall outside the scope of our work.

7.4.4. Non-Defesive State Indicators. We designed AppJitsu to detect app resiliency indicators. However, AppJitsu would observe UI layout differences based on other detection mechanisms as well. For instance, it is possible for an app to detect a resource (e.g., SIM card), which is directly related to the operation of an app, and display an error accordingly. In such rare cases, AppJitsu may evaluate the app-state difference as an indicator of a self-defense mechanism.

7.4.5. Custom Permission Requests. AppJitsu's *App-State Manager* module can handle permission requests through the UI layout hierarchy thanks to the centralized access control system in Android (§4.2.3). However, during our evaluation, we discovered that some apps present


Location Services Disabled

Allow "U" to access your location even when you are not using the app?

Your location is used to provide more relevant and accurate information.

Location permission

Knowing your location we can protect you against possible fraud from anywhere. You can also receive more personalized services.

Figure 8: Custom permission dialogs from `com.bbt.myfi` (left) and `es.bancosantander.apps` (right) apps. Custom pop-ups appear before a standard system dialog to ask for permissions, and hinders the permission-granting activity of AppJitsu.

a pop-up notification that the user needs to dismiss before they can grant permissions. Since the developers can construct the UI layout arbitrarily, there are no standard methods to detect and dismiss this notification [11]. In such cases, AppJitsu is limited to the app-state *before* we grant permissions. Consequently, any app which relies on a method, which is a part of a permission-controlled standard Android API, to deploy self-defense methods would fail to detect the hostile environment. A prominent example of such a case is the permission to make phone calls, which also gives access to fields that can reveal the emulator-specific strings. We present an example of two custom pre-permission request windows in Figure 8.

8. Related Work

Earlier works which evaluated Android apps at a large-scale [42], [43] focused on malware detection and analysis. Similarly, our previous large-scale work Libspector [44] identified different types of library usage in top Android apps, whereas BorderPatrol [45] demonstrated protection against malicious libraries at-scale. However such large-scale studies did not provide an understanding on the state of resiliency against app and runtime environment tampering in benign apps. As a response, researchers studies the attacks on Android app integrity. One of the early works is Protsenko et al's [46], which found 97% of top paid apps were susceptible to repackaging attacks. As a response, authors have built a native self-protection for tamper proofing Android apps. Unfortunately, their work provides protection against the repackaging, debugger and reverse engineering tools with limited scope.

For a wider understanding of how multiple resiliency methods are in place, researchers also analyzed more than one attack vector at a time. Haupert et al, [14] examined a widely used library which provides app self-protection, and demonstrated two runtime attacks against the protections in place to disable security measures. In their work, they analyzed the custom libraries which can provide multiple self-protection methods, however were able to exploit the integrity of apps regardless. More related to our work is by Berlato and Ceccato. [11], where authors statically analyzed the presence and adoption of anti-debug and anti-tampering code in Google Play apps from 2015 and 2019.Their insights showed a decreasing popularity for anti-debug and anti-tampering methods. Their work showed decreasing adoption of propriety anti-tamper library usage across years, and limited use of the Google-provided SafetyNet Attestation API, which can check the runtime environment integrity. Unfortunately, their system relies on static analysis and does not span over the entire OWASP resiliency requirements which are related to detection of tamper methods.

In response to the growing attacks to financial industry, researchers studied potential vulnerabilities in banking apps. Nguyen-Vu et al. [12], examined the root detection and anti-root evasion techniques. They surveyed 110 root checking apps and the implementation of root checking methods, then evaluated 28 thousand Android apps (including 7200 malware samples) to see if such methods are in place. Although a comprehensive study in Android rooting, their work falls short on coverage of the OWASP resiliency requirements. Similar to our work, Phumkaew and Visoottiviseth [47] analyzed hospital and stock trade applications from Thailand and extracted data-at-rest from mobile devices with `adb` to demonstrate importance of OWASP Top 10 mobile threat analysis. However, their work is limited to modifying app packages and using a rooted device for code tamper-detection, which only satisfies two of the resiliency requirements. Another work by Kim et al. [13] identifies API calls to check device rooting and app integrity. They examined 76 popular financial Android apps in the Republic of Korea, and then devised methods to bypass mechanisms of five libraries which provide self-defense methods. Another static analysis by Chen et al. [48] scrutinized banking app packages to detect weaknesses in input/output structures, data storage and sensitive data transmission. Authors of STAMBA [49] created a framework to test mobile banking apps in terms of the secure communication requirements of apps, however they did not study anti-tampering requirements.

Finally, UI-driven app testing has been a method of choice in earlier works [16]–[20]. Similar to an earlier work of Bianchi et al [32], our app-state indicator also uses *uiautomator* to control the Android UI and use perceptional hash on the screen layout hierarchy.

9. Conclusions

In this paper, we designed and built AppJitsu, a dynamic app analysis framework that evaluates the resiliency in security-sensitive apps. Using our AppJitsu prototype, we analyzed the most popular 455 `FINANCE` apps from the Google Play on multiple systematically-constructed hostile runtime environments. We then presented our implementation on how AppJitsu detects indicators of resiliency in Android apps, as well as our data analysis methodology methodology. Finally, we demonstrated the lack of self-defense methods in popular finance-related apps, and studied the manifestation of each specific resiliency indicator in their respective runtime environment configuration. Our results indicate that 25.05% of the tested apps lack all recommended self-defense mechanisms, whereas only 10.77% employed *all* defensive methods we tested. In conclusion, AppJitsu determined that nearly one fourth of Financial apps do not employ *any* defense at all, while only a small fraction demonstrates resiliency against commonly known attack methods.

[11] Users can only grant permissions through standard system permission dialogs or system settings. Custom permission request windows serve as informational messages.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and feedback. This material is based upon work supported by the NSF under award numbers CNS-1949632 and CNS-1942793, and the ONR under grant number N00014-19-1-2364. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views, either expressed or implied, of the funding agencies.

References

[1] O. Design, "Mobile Ecommerce Statistics." https://www.outerboxdesign.com/web-design-articles/mobile-ecommerce-statistics, 2020.

[2] B. Insider, "Business Insider Intelligence Report." https://www.businessinsider.com/intelligence/research-store/The-Mobile-Checkout-Report/p/58319012, 2018.

[3] TheHill, "FBI warns hackers are targeting mobile banking apps." https://thehill.com/policy/cybersecurity/502148-fbi-warns-hackers-are-targeting-mobile-banking-apps, 2020.

[4] RSA, "RSA Mobile App Fraud Report." https://www.rsa.com/en-us/blog/2018-05/rsa-fraud-report-mobile-app-fraud-/transactions-increased-over-600-percent-in-three-years, 2018.

[5] SecurityIntelligence, "IBM Trusteer exposes massive fraud operation facilitated by evil mobile emulator farms." https://securityintelligence.com/posts/massive-fraud-operation-evil-mobile-emulator-farms/, 2020.

[6] I. S. Trusteer, "The New Frontier of Fraud – Massive Mobile Emulator Fraudulent Operation ." https://community.ibm.com/community/user/security/blogs/doron-hillman1/2020/12/23/trusteer-new-frontier-of-fraud-mobile-emulator, 2020.

[7] Google, "Android Security Tips." https://developer.android.com/training/articles/security-tips, 2020.

[8] OWASP, "Mobile Top 10 Risks." https://owasp.org/www-project-mobile-top-10/, 2016.

[9] OWASP, "Mobile AppSec Verification Standard." https://github.com/OWASP/owasp-masvs, 2020.

[10] Google, "SafetyNet Attestation API." https://developer.android.com/training/safetynet/attestation, 2020.

[11] S. Berlato and M. Ceccato, "A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps," *Journal of Information Security and Applications*, vol. 52, p. 102463, 2020.

[12] L. Nguyen-Vu, N.-T. Chau, S. Kang, and S. Jung, "Android rooting: An arms race between evasion and detection," *Security and Communication Networks*, vol. 2017, 2017.

[13] T. Kim, H. Ha, S. Choi, J. Jung, and B.-G. Chun, "Breaking ad-hoc runtime integrity protection mechanisms in android financial apps," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 179–192, 2017.

[14] V. Haupert, D. Maier, N. Schneider, J. Kirsch, and T. Müller, "Honey, i shrunk your app security: The state of android app hardening," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 69–91, Springer, 2018.

[15] GuardSquare, "ProGuard." https://www.guardsquare.com/en/products/proguard, 2020.

[16] S. Chen, L. Fan, G. Meng, T. Su, M. Xue, Y. Xue, Y. Liu, and L. Xu, "An empirical assessment of security risks of global android banking apps," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1310–1322, 2020.

[17] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 23–26, IEEE, 2017.

[18] Y. Ma, Y. Huang, Z. Hu, X. Xiao, and X. Liu, "Paladin: Automated generation of reproducible test cases for android apps," in *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pp. 99–104, 2019.

[19] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 224–234, 2013.

[20] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 245–256, 2017.

[21] Google, "UI Automator testing framework." https://developer.android.com/training/testing/ui-automator, 2020.

[22] Google, "Google Play." https://play.google.com/store.

[23] OWASP, "Mobile Top 10 Risks: M8:Code Tampering." https://owasp.org/www-project-mobile-top-10/2016-risks/m8-code-tampering, 2016.

[24] OWASP, "Mobile Security Testing Guide." https://github.com/OWASP/owasp-mstg, 2020.

[25] Google, "Android Debug Bridge." https://developer.android.com/studio/command-line/adb, 2020.

[26] S. Margaritelli, "Smali Emulator." https://github.com/evilsocket/smali_emulator, 2016.

[27] Frida, "Frida Dynamic Instrumentation Toolkit." https://frida.re/, 2020.

[28] Google, " Run ARM apps on the Android Emulator ." "https://android-developers.googleblog.com/2020/03/run-arm-apps-on-android-emulator.html, 2020.

[29] rovo89, "Xposed Framework API." http://api.xposed.info/reference/packages.html, 2020.

[30] J. Wu, "Elder driver Xposed Framework." https://github.com/ElderDrivers/EdXposed, 2020.

[31] Google, "ADB Monkey UI Exerciser." https://developer.android.com/studio/test/monkey.html, 2020.

[32] A. Bianchi, E. Gustafson, Y. Fratantonio, C. Kruegel, and G. Vigna, "Exploitation and mitigation of authentication schemes based on device-public information," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, pp. 16–27, 2017.

[33] xiaocong, "UIautomator: Python wrapper of Android uiautomator test tool." https://github.com/xiaocong/uiautomator, 2020.

[34] J. Buchner, "Python perceptional image hashing module." https://github.com/JohannesBuchner/imagehash, 2020.

[35] AndroidRank, "Most downloaded free Android applications." https://www.androidrank.org/app/ranking/all?sort=4&price=free, 2019.

[36] R. Alam, "gplaydl: Commandline Google Play APK downloader." https://github.com/rehmatworks/gplaydl, 2020.

[37] C. Tumbleson, "apktool." https://github.com/iBotPeaches/Apktool, 2020.

[38] Google, "Android asset packaging tool." https://developer.android.com/studio/command-line/aapt2, 2020.

[39] Splitwise, "Splitwise app." https://play.google.com/store/apps/details?id=com.Splitwise.SplitwiseMobile&hl=en_us, 2020.

[40] I. R. Service, "IRS2Go app." https://play.google.com/store/apps/details?id=gov.irs&hl=en_us, 2020.

[41] H. Zhou, T. Chen, L. Haoyu Wang, Yu, X. Luo, T. Wang, and W. Zhang, "Ui obfuscation and its effects on automated ui analysis for android apps," in *35th IEEE/ACM International Conference on Automated Software Engineering*, 2020.

[42] M. Bierma, E. Gustafson, J. Erickson, D. Fritz, and Y. R. Choe, "Andlantis: Large-scale Android dynamic analysis," *arXiv preprint arXiv:1410.7751*, 2014.

[43] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer, “Andrubis–1,000,000 apps later: A view on current Android malware behaviors,” in *Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, pp. 3–17, IEEE, 2014.

[44] O. Zungur, G. Stringhini, and M. Egele, “Libspector: Context-aware large-scale network traffic analysis of android applications,” in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 318–330, IEEE, 2020.

[45] O. Zungur, G. Suarez-Tangil, G. Stringhini, and M. Egele, “Borderpatrol: Securing byod using fine-grained contextual information,” in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 460–472, IEEE, 2019.

[46] M. Protsenko, S. Kreuter, and T. Müller, “Dynamic self-protection and tamperproofing for android apps using native code,” in *2015 10th International Conference on Availability, Reliability and Security*, pp. 129–138, IEEE, 2015.

[47] N. Phumkaew and V. Visoottiviseth, “Android forensic and security assessment for hospital and stock-and-trade applications in thailand,” in *2018 15th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pp. 1–6, IEEE, 2018.

[48] S. Chen, L. Fan, G. Meng, T. Su, M. Xue, Y. Xue, Y. Liu, and L. Xu, “An empirical assessment of security risks of global android banking apps,” *arXiv preprint arXiv:1805.05236*, 2018.

[49] S. Bojjagani and V. Sastry, “Stamba: Security testing for android mobile banking apps,” in *Advances in Signal Processing and Intelligent Recognition Systems*, pp. 671–683, Springer, 2016.