MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Methods for persistence in Android Malware

Bachelor's Thesis

PATRÍCIA GORCOVÁ

Brno, Spring 2025

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Methods for persistence in Android Malware

Bachelor's Thesis

PATRÍCIA GORCOVÁ

Advisor: RNDr. Lukáš Němec, Ph.D.

Department of Computer Systems and Communications

Brno, Spring 2025


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source. During the preparation of this thesis, I used the following AI tools:

- Grammarly for grammar correction
- ChatGPT for grammar correction and stylistic control

Patrícia Gorcová

Advisor: RNDr. Lukáš Němec, Ph.D.

Acknowledgements

Big thank you to my advisor Lukáš Němec and my family and also thank you for reading this.

Abstract

Persistence is a critical feature of modern Android malware, enabling long-term control over infected devices and resilience against user-initiated removal, reboots, and even factory resets. This thesis presents a structured categorization of persistence techniques employed by Android malware, grouped according to the system components they compromise, from firmware-level image tampering and system library injection to user-space mechanisms like receivers, services, schedulers, and UI/UX manipulation. Each category is examined with technical detail, real-world malware examples, and a focus on operational implications. The analysis is complemented by a practical demonstration: a persistent application that survives a factory reset and runs automatically at boot. This prototype, deployed on a rooted physical device, simulates malware-like behavior and highlights key challenges of real-world persistence implementation.

Keywords

Android, persistence, malware, rootkits, Triada, Verified Boot (AVB)

Contents

Introduction **1**

1 Android OS and Android malware **2**
1.1 Android Partitioning 3
1.2 Android Boot Process 5

2 Techniques of Malware Persistence on Android **7**
2.1 Android Image Tampering/Compromise 8
2.2 Abusing services 17
2.3 UI/UX manipulation 20
2.4 Abusing receivers 22
2.5 Abusing system schedulers 23
2.6 Summary 25

3 Case Study of a Persistent Malware: *Triada* **27**
3.1 Initial Infection and Delivery 27
3.2 Persistence Techniques 27
3.3 Capabilities and Behavior 28
3.4 Attribution and Motivation 29

4 Implementation **30**
4.1 Implemented Techniques 30
4.2 Tools and Device Setup 30
4.3 Development 31
4.4 Results 33
4.5 Issues and limitations 33

Conclusion **35**

Bibliography **37**

A An appendix **43**

List of Tables

1.1 Overview of Android firmware partitions 5
2.1 Summary of Android malware persistence techniques . . 26
3.1 Timeline of Triada's Evolution 29

List of Figures

2.1 Overwriting of the `install-recovery.sh` by SnapPea Downloader. 11
2.2 Service definition injected into `init.rc` by the OldBoot malware. 12
2.3 Command used by Pegasus to disable automatic system updates via `settings` CLI. 13
2.4 Generally used shell commands for installing a malicious APK as a system application. 17
2.5 BouldSpy's overriding of the `onDestroy()` method. 19
2.6 Manifest declaration for a BroadcastReceiver that listens for the `BOOT_COMPLETED` event. 23
2.7 Example of scheduling by using the `TimerTask` 25

Introduction

The Android operating system has become the dominant platform in the mobile computing landscape, powering billions of smartphones, tablets, and embedded devices worldwide. Its open-source nature, extensive device compatibility, and customizable architecture have made it a popular target not only for developers and manufacturers but also for malicious actors seeking to exploit its flexibility for nefarious purposes.

Among the objectives of modern Android malware is the ability to maintain a persistent presence on the infected device. Persistence ensures that the malware survives user-initiated countermeasures such as application removal, system reboots, or even factory resets. Without a persistent foothold, malware risks losing its influence over a device and being rendered ineffective in achieving its goals, whether those involve surveillance, data theft, financial exploitation, or long-term espionage.

Persistence on Android can be achieved through a wide variety of techniques, each leveraging different components of the Android software stack — from user-space application abuse, to system-level manipulations, and even low-level firmware compromises. These methods vary significantly in complexity, stealth, privilege requirements, and resilience against detection and mitigation strategies.

The goal of this thesis is to explore the techniques used by Android malware to achieve persistence. First, relevant background on Android architecture and malware evolution is presented. The core of the thesis offers a structured categorization of persistence techniques based on the technical layer of the system that they affect. Each method is described in terms of its prerequisites, technical implementation, and historical or contemporary use by real-world malware.

Finally, the thesis includes a practical implementation of one selected persistence technique, showcasing how an application can be installed as a system app and triggered at boot to simulate persistent behavior. The implementation highlights the technical steps involved, tools used, and challenges encountered, reinforcing the broader theoretical discussion with an applied example.

1 Android OS and Android malware

The Android operating system, developed by Android Inc. and later acquired by Google in 2005, quickly attracted a large community of developers and hardware manufacturers, due to its open-source ecosystem and flexible licensing model. Its first commercial version, Android 1.0, was released in September 2008. The system already came preloaded with several key Google services, including Google Maps, YouTube, and an HTML browser. Additionally, Android 1.0 introduced the first version of the Android Market, the predecessor of Google Play Store, that allowed developers to quickly distribute their apps with almost no setbacks[1].

By the end of 2012, Android had surpassed iOS in global smartphone sales, establishing itself as the dominant mobile operating system. As of early 2025, Android holds approximately 70% of the global mobile OS market share[2]. Its widespread adoption, open development environment, and broad support for applications created outside of Google's official development teams have made it not only the most popular but also one of the most targeted platforms by malicious actors.

Malware targeting Android devices emerged almost in parallel with the platform's growing popularity. The first known Android trojan, DroidSMS, appeared in 2010. Disguised as a simple media player, it secretly sent premium-rate SMS messages, leading to significant financial losses for unsuspecting users. To this day, nearly all known Android malware aims to generate revenue through illicit methods. While the evolution of Android malware mirrors trends observed in desktop environments, it has progressed at a significantly accelerated pace. In contrast, desktop malware underwent decades of technical innovation before financial motives became dominant; Android malware focused on profit or user data from the beginning. As Han et al. observe, this makes Android malware particularly notable for its ongoing exploration of the diverse ways smartphones can be exploited for large-scale monetization, which are often unavailable or impractical on traditional desktop platforms[3, p. 63].

With users increasingly relying on their phones for sensitive activities such as online banking, communication, health monitoring, and

secure authentication, mobile devices began to store vast amounts of valuable personal and financial data. The Android malware landscape evolved rapidly to exploit these opportunities. Early threats focused on SMS fraud and adware, but as mobile capabilities expanded, malware authors introduced more sophisticated threats such as banking trojans (e.g., Anubis, Cerberus) and advanced spyware like FlexiSpy and SpyNote. Malware campaigns increasingly targeted financially active users, business professionals, and government employees, whose devices contained high-value data. Over time, the strategic importance of smartphones also attracted the attention of state-sponsored devices and advanced persistent threat (APT) groups. High-profile espionage tools such as Pegasus demonstrated how mobile devices could be silently compromised to conduct surveillance on journalists, political figures, and activists. In this evolving landscape, Android malware transitioned from simple scams to sophisticated espionage tools, data exploitation, and geopolitical influence.

Regardless of the specific type of malware or motivation behind it, all adversaries ultimately share the same objective: to maintain a foothold on compromised devices for as long as possible while remaining undetected. Over the years, this goal has driven the continuous development and refinement of sophisticated persistence techniques, allowing malware to evade removal efforts, survive device reboots, and persist even through factory resets and system updates.

In order to understand how infections operate within the Android ecosystem, it is essential to first explore the architecture and Boot Process of the Android OS.

1.1 Android Partitioning

An Android device's firmware is composed of multiple vendor-supplied system images, each responsible for a specific role in the system's boot and runtime functionality. These images are flashed to designated storage partitions and serve distinct roles in the boot process and system operation.

The **Bootloader**, stored in the `aboot` or `hboot` partition, depending on the manufacturer, contains the initial bootstrap code responsible

for initializing hardware components, loading the boot image, and facilitating firmware updates.

The **Boot Image**, located in the `boot` partition, is a critical Android image that consists of the Linux kernel and a compressed RAM disk. During a normal boot, the RAM disk serves as the initial root filesystem, and its `init.rc` scripts define how the remaining system components and services are loaded.

The **Recovery Image** is stored in the `recovery` partition. This is a specialized minimal functioning Android image that also contains a kernel and RAM disk, but is used to boot the device into recovery mode for maintenance tasks such as firmware updates or system repairs when normal booting fails.

The **System Image** resides in the `system` partition. This image holds the core Android operating system, including system libraries, frameworks, and pre-installed applications provided by Google, the device manufacturer, and network carriers.

The **Data Partition** contains user data and system configuration files. It also defines the device's factory default state, which is restored during a factory reset operation.[4, p. 62]

The **Vendor Partition** contains device-specific drivers and hardware abstraction layer (HAL) components provided by the device manufacturer.

There are also other partitions like **misc**, **cache**, and others used for miscellaneous configuration data, cached files, etc.[5]

Table 1.1: Overview of Android firmware partitions

Name	Mount Point	Root / Permissions	Short Description
`boot`	`/aboot` or `/hboot`	Yes / Read-only	Contains the Linux kernel and RAM disk. Used to boot the main operating system.
`recovery`	`/recovery`	Yes / Read-only	Minimal recovery OS and kernel used for firmware updates and factory resets.
`system`	`/system`	Yes / Read-only	Holds the Android OS core: libraries, frameworks, and pre-installed system apps.
`vendor`	`/vendor`	Yes / Read-only	Contains device-specific drivers and HALs provided by the OEM.
`data`	`/data`	No / Read-write	Stores user data, installed apps, and settings. Wiped during factory reset.
`cache`	`/cache`	No / Read-write	Temporary system data, used for tasks like OTA updates and logs.
`misc`	Varies	Yes / Varies	Holds low-level system settings such as boot mode flags or carrier configurations.

1.2 Android Boot Process

The Android boot process is a multi-stage sequence that initializes device hardware, verifies system integrity, and launches the operating system. The key stages are as follows:

1. Boot ROM: Embedded in the device's hardware, this immutable code initializes the hardware and loads the next stage—the bootloader.

2. Bootloader: A vendor-specific program responsible for verifying and loading the Android boot image. If security features such as Verified Boot are enabled, the bootloader performs cryptographic integrity checks before proceeding.

3. Boot Image ('boot.img'): Contains the Linux kernel and a compressed ramdisk. The ramdisk includes the 'init' process and associated initialization scripts ('init.rc'), which are critical for mounting partitions and starting essential system services.

4. init Process and Zygote: The 'init' process is the first user-space process executed by the kernel. It initializes system properties

and services, including the launch of the Zygote process. Zygote is responsible for spawning all Android application processes.

5. System Initialization: The Android runtime environment (ART), system services, and application frameworks are loaded, completing the transition to a fully operational state.

2 Techniques of Malware Persistence on Android

Persistence is a critical feature of advanced Android malware, allowing malicious actors to maintain long-term access to compromised devices despite user attempts at removal. Adversaries may implement this feature in multiple ways. These techniques vary significantly in their implementation complexity, stealth, and resistance to mitigation. Usually, many of these techniques are used simultaneously in malware to prolong its presence and runtime on the compromised device for as long as possible.

The persistence techniques presented in this thesis are categorized based on the technical layer of the Android system that the malware modifies or exploits in order to survive reboot, removal, or factory reset. This classification approach reflects the depth of system penetration and the associated resistance to detection and remediation. It should be noted, however, that the effectiveness of detection also depends on additional factors such as the sophistication of the malware's obfuscation techniques and the presence of security mechanisms on the device. Each main category in this chapter corresponds to a distinct technical vector, and the subsections within each category detail the various sub-techniques used, their prerequisites, and real-world malware examples.

It's essential to recognize that there are different methodologies for categorizing persistence techniques. One of the most well-known frameworks is the MITRE ATT&CK® for Mobile[1], which classifies adversarial behaviors according to their tactical objectives. The persistence tactic approach focuses on when and how malicious code re-executes over time, rather than on which specific system layer (firmware, system, user-space) is modified. This allows the framework to stay focused on observable adversary behaviors and techniques. While this model is highly valuable for threat intelligence and incident response, the approach adopted in this thesis is specifically designed to emphasize the technical infection vectors and affected system components. This perspective is particularly relevant for un-

1. See `https://attack.mitre.org/tactics/mobile/`

derstanding the practical implementation of persistence methods and their implications for system security and forensic analysis.

2.1 Android Image Tampering/Compromise

One of the most potent and resilient methods of achieving malware persistence on Android devices involves tampering with critical system images. This approach typically targets low-level components such as the boot, recovery, or system partitions. By compromising these elements, attackers can inject malicious code that persists through device reboots and factory resets, and gain the ability to execute code during privileged stages of the boot process.

The origins of image tampering as a persistence strategy can be traced back to the early years of Android malware development. Between 2010 and 2014, the technique of installing malware into system directories gained popularity due to the relative ease of obtaining root access and the absence of robust integrity verification mechanisms. Malware authors frequently leveraged public rooting toolkits such as Framaroot[2] to gain superuser privileges and place their payloads into protected system directories. According to Kaspersky, this marked the beginning of a dangerous trend: many malware families began exploiting root access automatically, without user interaction, granting them virtually unrestricted control over the device while bypassing Android's application sandboxing model[6].

However, beginning around late 2016, this trend began to decline. The introduction of Verified Boot[3] in Android 7.0 enforced cryptographic integrity checks during the boot process, significantly complicating unauthorized modifications to system images. Additionally, the reduced availability of public root exploits and the growing diversity of hardware configurations made the development of reliable, universal rooting payloads increasingly impractical. Consequently, many malware developers began shifting their focus to user-space persis-

2. See `https://www.framaroot.net/`

3. In earlier Android versions, Verified Boot warned users about detected image corruption but still allowed the device to boot. For more information, see `https://source.android.com/docs/security/features/verifiedboot`

tence techniques, such as abusing Accessibility Services or foreground service mechanisms.

As discussed, image tampering requires privileged access to the device's firmware partitions. This level of access can typically only be achieved through:

- Exploiting vulnerabilities that provide root or bootloader-level privileges.
- Gaining physical access to the device and using specialized tools (e.g., flashing via Fastboot[4] or JTAG interfaces).
- Conducting insider supply chain attacks, in which system images are modified during manufacturing or by third-party vendors before the device reaches the end user.

Despite modern advancements in Android security, firmware-level persistence remains relevant in specific high-stakes threat scenarios, particularly those involving espionage, targeted surveillance, or nation-state operations. In such contexts, adversaries may still possess the necessary capabilities, including access to leaked OEM signing keys, zero-day vulnerabilities, or physical manipulation of devices, to tamper with firmware images without detection. While mechanisms like Verified Boot have substantially improved resilience, they are not absolute barriers against determined and well-resourced attackers.

Recovery install script patching

The Android recovery environment is a separate runtime partition, typically loaded through the `recovery.img` image, designed for system maintenance tasks, such as factory resets, firmware updates, and system repairs. It operates independently of the main Android OS, providing a minimalistic interface with elevated privileges.

Recovery may update `/boot` or `/system`, but it never overwrites itself. The `/system/etc/install-recovery.sh` script is automatically executed during boot to perform two primary functions: it verifies the integrity of the recovery partition, ensuring that the recovery environment matches the expected official version, and if any unauthorized

4. See `https://source.android.com/docs/setup/test/running`

modifications are detected, such as the installation of custom recoveries like Team Win Recovery Project (TWRP)[5], it restores or updates the recovery partition by reflashing the official stock recovery image. It also typically reflashes the recovery at the first boot after a factory reset. This mechanism was designed to maintain the device's security posture by preventing users from installing custom firmware or gaining privileged system access through modified recovery environments. However, a compromise of this script means that malware can execute arbitrary code during every boot or recovery event, ensuring its persistence even after factory resets and system recovery operations.

SnapPea Downloader: One of the malware that used this method was SnapPea Downloader in 2015, which affected Android versions 4 and 5. It is a part of the GhostPush malware family from which a famous variant named Gooligan later emerged. The infection process was initiated when a user installed a malicious version of the SnapPea Windows backup application on their PC. Upon connecting an Android device to the infected PC, the compromised SnapPea application installed its Android component onto the device with the user's approval. This component then deployed repackaged versions of seemingly legitimate apps containing embedded malicious payloads. These apps used a suite of twelve known vulnerabilities to gain root access (two of which have assigned CVEs, VROOT – CVE-2013-6282 and Towelroot – CVE-2014-3153), thereby granting the malware complete control over the system[7].

In case of successful privilege escalation, the application runs an installation shell script. Below in the Figure 2.1. excerpt from the script focusing on the `install-recovery.sh` file modification.

5. See `https://twrp.me/about/`

```
# first remounts /system as read-write and copies malware
    files to /system/xbin/.ext.base and to /system/xbin/.b.

# remove immutable and append-only attributes from the
    recovery script if set
chmod -i -a /system/etc/install-recovery.sh;

# append to the recovery script to execute malware binary on
    recovery/startup
echo -e '#!/system/bin/sh\n /system/xbin/.ext.base  & \n '  >
    /system/etc/install-recovery.sh;

# set executable permissions on the recovery script
chmod 755 /system/etc/install-recovery.sh;

# make the recovery script immutable and append-only again
chattr +i +a /system/etc/install-recovery.sh;
```

Figure 2.1: Overwriting of the `install-recovery.sh` by SnapPea Downloader.

Other examples: During the upcoming years, there have been a few malware samples that used this mechanism to maintain persistence. The **Rootnik** malware family, first seen in 2015, modifies the `install-recovery.sh` in multiple variants over the years[8, 9]. Another example is an app named **Cloud Module**, discovered in 2018, whose main function was stealing data[10]. Perhaps the most famous one using this technique became known as the unkillable **xHelper** in 2019[11].

Modifying `init.rc`

The `init.rc` script is part of Android's init process, which is the first user-space process executed by the Linux kernel after boot. This script is responsible for initializing system properties, mounting essential filesystems, starting core system daemons, and setting up critical services. Any modifications to this script directly affect the behavior of the system during boot, making it an attractive target for attackers seeking to establish early and persistent execution of their payloads. By introducing additional service definitions or command executions

into `init.rc`, attackers can ensure that their malicious binaries are executed automatically upon every device boot.

OldBoot: A classic example of this technique is the first Android bootkit discovered in the wild, OldBoot. This malware was also one of the first instances of pre-installed malware on Android devices[3, p. 81]. The 2014 variant `Android.OldBoot.1` modified the script to insert a new malicious service and associated socket configuration. It creates a service that runs `/sbin/imei_chk`, which acts as an installer for the malware[12].

```
# create service which executes the playload
service imei_chk /sbin/imei_chk

# start service during the early core boot phase
class core

# create socket that allows other processes to communicate
    with the service
socket imei_chk stream 666
```

Figure 2.2: Service definition injected into `init.rc` by the OldBoot malware.

Update supression

Update suppression is a sophisticated persistence technique employed by advanced Android malware to prevent the operating system from receiving critical security patches and updates. By obstructing system updates, malware ensures that vulnerabilities remain unpatched, allowing the malicious software to maintain its foothold on the device and continue its operations without interruption.

Malware can suppress system updates through various methods:

- Malware can disable or corrupt Android system services responsible for managing Over-The-Air (OTA) updates, such as `update_engine`[6].

6. See `https://android.googlesource.com/platform/system/update_engine/`

- Automatic updates also play an important role. Malware can turn off the Android system's automatic update functionality by modifying the system settings, reducing the likelihood that the user or device would apply the updates.

Pegasus: A high-end surveillance platform developed by the NSO Group, used by many governments, showcases the use of update suppression as part of its sophisticated persistence strategy. Pegasus, or rather the Android variant named **Chrysaor**, is a notable example of malware that employs update suppression as part of its persistence strategy.

The variant, which was described closely by Lookout, disabled the automatic software updates by setting `SOFTWARE_UPDATE_AUTO_UPDATE` to 0. This setting is not part of the official Android SDK or documented in the Android Open Source Project (AOSP)[7]. It is probably a system setting utilized by certain manufacturers, such as Samsung, to control automatic system updates. There was also a component tied to specifically targeting Samsung devices; the `com.sec.android.fotaclient` Samsung's system updater, which Pegasus managed to remove[13, 14].

```
# the turning off uptades could have been done using the
    /system/bin/settings
settings put system SOFTWARE_UPDATE_AUTO_UPDATE 0

# removing the system updater
mount -o remount,rw,execsuid /system
rm /system/app/FotaClient.apk
rm /system/app/FotaClient.odex
pm disable com.sec.andoid.fotaclient
```

Figure 2.3: Command used by Pegasus to disable automatic system updates via `settings` CLI.

7. See `https://developer.android.com/reference/`

System Library and Binary Injection/Hijacking

Android applications and services rely heavily on shared libraries, typically written in C or C++ and packaged as `.so` (Shared Object) files. These libraries are loaded by system processes and applications at runtime to provide core functionality such as cryptographic operations, network communication, and media handling. These libraries are loaded by system processes and applications at runtime to provide core functionality such as cryptographic operations, network communication, and media handling. They are placed in `/system/lib/` or `/system/lib64/`[4, p. 42].

System binaries, on the other hand, are executable files that implement essential system commands and services (e.g., `netd`, `servicemanager`, or custom OEM binaries). These binaries often run with elevated privileges and are critical to system stability. These are located in `/system/bin/`, `/system/xbin/`[4, p. 12, 13].

There are multiple ways in which this technique can be achieved and implemented:

1. Creating or replacing: Attackers can create new libraries or binaries in `/system` directory with malicious code or replace legitimate files with modified versions that include malicious code. This is especially viable for supply-chain attacks where the malicious library is replaced or added by the vendor or supplier during the manufacturing process.

A prime example is the **Triada** malware, later described more closely as a case study in Chapter 3, which modified the system library `libandroid_runtime.so`. Next, the already mentioned **OldBoot** bootkit, among other installed a library `libgooglekernel.so` into `/system/lib`[12]. The **SnapPea Downloader** also implemented this technique, and the installation script also copied two files into `/system/xbin`[7]. **xHelper** places files `/system/xbin` and runs them from `install-recovery.sh`, thus effectively combining the two techniques[11].

2. Patching: After the malware is installed, attackers can patch system binaries and libraries to inject malicious shell commands or code that is executed when critical services start, like modifying `init` or

`servicemanager` binaries to automatically launch malware components during system initialization or injecting into `libandroid_runtime.so` malware ensures that its payload is loaded automatically by system processes or user apps during normal operation.

The first documented malware to use this technique was **Dvmap** in 2017. It managed to inject code into system libraries `libdmv.so` and `libandroid_runtime.so` during runtime[15].

3. Function Hooking and API Hijacking: Instead of replacing entire libraries or injecting right into them, attackers can hook into specific exported functions by modifying the library's symbol table or replacing frameworks. This allows malware to intercept and manipulate system API calls without requiring full library replacement.

This method can be seen with the **Zen** malware in 2019, which can replace `framework.jar`, which allows it to modify the Android API[16].

4. Process Injection: Some malware employs runtime-only code injection by manipulating already-running privileged processes using debugging interfaces like `ptrace`. This allows the injection of malicious libraries into memory, bypassing static filesystem modifications entirely. This means, however, that the library will *not be loaded across reboots or turn-offs* and the malware would need to employ another technique to ensure the library's loading at boot. Nevertheless, it demonstrates a highly stealthy form of in-memory persistence, usually targeting a highly privileged process, which is not possible to kill as a user, that avoids on-disk modifications.

A notable example is **Loki**, which targets the `system_server` process, a core Android component forked directly from Zygote. Loki attaches to the process using `ptrace()`, locates writable memory via `/proc/self/maps`, and dynamically injects its payload (`liblokih.so`) into the process's address space. This is functionally equivalent to `LD_PRELOAD` abuse on Linux, but adapted for Android's runtime constraints[17, 18]. Other notable malware using this technique is **HummingBad**[19, 20].

Installing as a System Application

Another highly effective and widely used persistence technique in all types of Android malware is the installation of malicious applications directly into protected system directories, such as `/system/app/` and `/system/priv-app/`. By placing malware in these locations, attackers ensure their applications are loaded during system startup and persist across reboots.

- `/system/app/`: This directory is used for standard system applications installed by device manufacturers or during the initial firmware setup. Applications placed here are treated as trusted system apps but still adhere to certain permission limitations. These include the prebundled apps from Google, as well as any vendor or carrier-installed apps (though these should technically reside in /vendor/app, instead).
- `/system/priv-app/`: Applications in this directory have significantly more privileges. They can request and be automatically granted permissions not available to regular apps (e.g., access to SMS, call logs, device identifiers), and their activities are less scrutinized by Android's permission system. This is why many core system services and critical apps reside in this directory.

BusyGasper: Discovered in 2018, BusyGasper was a sophisticated spyware platform with espionage capabilities typically associated with advanced persistent threat (APT) actors. The malware targeted a limited number of Android devices, likely in a highly targeted surveillance campaign, with fewer than 10 victims discovered. The malware consisted of multiple components, where the first one had installed other modules into the `/system/priv-app`[21].

Other examples: From the already mentioned malware, **Oldboot** leveraged this technique as well. Then in 2015, a new malware called **BrainTest** emerged, which used this technique, and installed a malicious application as a system app[22]. There is also **SpyDealer** although it was discovered in 2017, it still used the exploits for rooting targeted at devices with an Android version of 4.4 and lower[23].

Another sample leveraging this technique is **DoubleAgent**, discovered in 2020 but active since 2013, as a part of a highly sophisticated surveillance campaign[24].

Below is an example of how most of the mentioned malware installs the application into the protected partition.

```
# remount the /system partition with read-write permissions
mount -o remount,rw /system

# copy the malicious APK into the system app directory
cp malicious.apk /system/app/malicious/

# set the file ownership to the 'system' user and group
chown system.system /system/app/malicious/malicious.apk

# set appropriate file permissions (readable by system)
chmod 644 /system/app/malicious/malicious.apk

# enable the malicious application (registers it with the
    system)
pm enable com.android.malicious
```

Figure 2.4: Generally used shell commands for installing a malicious APK as a system application.

2.2 Abusing services

Services[8] are a fundamental Android component designed to perform background operations independently of a user interface. While they serve legitimate functions — such as media playback, data synchronization, and sensor monitoring — their flexible and persistent nature also makes them an attractive target for abuse by malware authors.

In early Android versions (prior to 8.0), services, especially background services, were largely unrestricted. Malware frequently exploited this by registering services that launched at boot and remained active indefinitely, enabling constant monitoring, logging, or remote control. However, in response to widespread abuse and battery drain

8. See `https://developer.android.com/develop/background-work/services`

concerns, Google introduced Background Execution Limits starting with Android 8.0 (API level 26). These restrictions prevent background services from running freely when the app is not in the foreground, unless they are promoted to foreground services.

As a result, modern malware has adapted by either:

- Using foreground services to bypass background restrictions, often with deceptive or invisible notifications;
- Using companion components, such as boot receivers and schedulers, to restart services on demand.

Despite these limitations, services remain a widely used and flexible vector for maintaining persistence.

Background Services

Background services are typically started through `startService()` and continue running in the background until manually terminated by the system or user. Malware can exploit background services to perform activities such as logging, data exfiltration, or state monitoring; activities that require constant execution but minimal visibility.

However, since the introduction of Doze Mode and App Standby in Android 6.0[9], and the stricter execution limits in Android 8.0+, background services face increasing system-imposed termination if the app is not actively in use. Malware authors can circumvent this by using the `START_STICKY` return flag in `onStartCommand()`, which instructs the system to automatically restart the service after it is killed.

BOULDSPY: A highly sophisticated surveillance platform, identified in 2023, demonstrates the abuse of Android services for persistent espionage operations. It launched background services that remained running across reboots and allowed for real-time microphone access, SMS surveillance, and remote file exfiltration.

The code below, found in BouldSpy's service implementation, overrides the `onDestroy()` method to prevent service termination. When

9. See `https://developer.android.com/training/monitoring-device-state/doze-standby`

the system or user attempts to stop the service, it broadcasts a custom intent and immediately restarts the core surveillance component. This ensures continued execution even under aggressive service management policies[25].

```
@Override
public void onDestroy() {
    super.onDestroy();
    this.wl.release(); // Release wakelock
    // Log destruction for debugging
    Log.d("BouldSpy", "onDestroy: Service being destroyed");
    // Create intent with custom action to broadcast restart
    Intent restartIntent = new Intent("YouWillNeverKillMe");
    this.getApplicationContext().sendBroadcast(restartIntent);
    // Explicitly restart the main surveillance service
    this.startService(new Intent(this, MainService.class));
}
```

Figure 2.5: BouldSpy's overriding of the `onDestroy()` method.

Foreground Services

To circumvent background restrictions on newer Android versions, malware increasingly uses foreground services, which are granted higher execution priority by the operating system. Foreground services must display a persistent notification to the user, but malware often disguises this notification using benign labels or transparent overlays to avoid detection.

The transition from background to foreground service abuse illustrates how Android's security hardening has shifted malware strategies from stealth to deception. Foreground services not only survive memory cleanup routines but also retain access to sensitive system APIs — such as the microphone, camera, and location services —even when the app is not active.

Mandrake: This spyware campaign has been active since 2016 and was first discovered in 2020; distributed via Google Play and third-party stores under the guise of utilities or lifestyle apps. It uses foreground services to maintain persistence while performing continuous

surveillance on infected devices. By displaying a transparent or misleading notification, the malware was able to stay active without alerting the user, leveraging Android's own APIs to justify its prolonged execution[26, 27].

2.3 UI/UX manipulation

User Interface (UI) and User Experience (UX) manipulation represents a class of malware techniques primarily aimed at defense evasion and, in some cases, achieving persistent execution through deception. While many of these techniques focus on hiding malicious applications from user awareness, thereby preventing manual termination or uninstallation, certain strategies actively simulate legitimate system behavior to mislead users and retain control over the device.

Predator: This is a mobile surveillance platform attributed to the commercial spyware vendor Cytrox and linked to highly targeted cyberespionage operations. One of its key techniques involves UI manipulation to simulate a device shutdown, allowing surveillance to continue even while the user believes the phone is powered off, Predator remains active in the background. This can ensure that the spyware components won't be killed by the system's shutdown or reboot[28].

Accessibility Services

Although technically implemented as a subclass of Android's Service component, Accessibility Services[10] represent a distinct and powerful attack vector due to their elevated privileges and direct interaction with the user interface. Originally introduced in Android 1.6 (Donut) to assist users with visual or motor impairments, Accessibility Services gained significantly more functionality beginning with Android 4.0 and especially 4.1 (Ice Cream Sandwich and Jelly Bean). These later improvements provided developers with the tools to create advanced assistive technologies, but also opened the door to abuse.

10. See `https://developer.android.com/guide/topics/ui/accessibility/service`

An app can register an Accessibility Service by extending the `AccessibilityService class` and declaring appropriate metadata in the `AndroidManifest.xml` and an XML configuration file. Once activated by the user, often through deceptive prompts or social engineering, the service is granted extensive access to system-wide UI events and interaction capabilities. These include:

- **Monitoring UI events:** Apps receive system-wide accessibility events, such as `TYPE_VIEW_CLICKED`, `TYPE_VIEW_TEXT_CHANGED` and `TYPE_WINDOW_STATE_CHANGED`, enabling observation of all user interactions..
- **Interacting with UI Elements:** Via the `AccessibilityNodeInfo` API, services can traverse the view hierarchy, click buttons, input text, and trigger other actions
- **Performing Global Actions:** Services can invoke `performGlobalAction()` to simulate pressing system buttons like Home, Back, or Recent Apps.
- **Reading on screen content:** Accessibility Services can extract text and labels from UI elements, potentially exposing passwords, SMS codes, and other sensitive information.

Once the user enables these services, malware can bypass permission dialogs, manipulate other apps' UIs, and grant itself new capabilities, all without direct user awareness. While all these actions are mainly useful for data exfiltration, they can be helpful in achieving persistence due. This technique is best described by showcasing it with an example below.

Mandrake: This malware offers a particularly instructive example of Accessibility-based persistence. Beyond surveillance and remote control, Mandrake employed a layered defense model using both the *Device Administrator* and *Accessibility* privileges to resist removal.

Mandrake gains Device Administrator access by using deceptive prompts styled as legitimate system requests (e.g., "enable battery optimization" or "activate advanced features"), manipulating the UI via Accessibility services. Once the user accepts, UI detection and

manipulation are once again used to intercept any attempts to disable either Device Administrator or Accessibility privileges, effectively blocking the user from accessing the appropriate settings screens or confirming removal dialogs, forming a closed loop of protection[26]. For typical users, the only reliable removal method is to boot into safe mode, manually revoke administrative privileges, and then uninstall the app.

2.4 Abusing receivers

Broadcast receivers[11] are a core Android component designed to respond to system-wide or application-specific broadcast messages, known as intents. These intents are emitted by the system in response to key lifecycle events such as device boot, app installation or removal, network changes, or user actions. By registering a BroadcastReceiver, either statically in the manifest or dynamically at runtime, an application can execute code in reaction to these events, without any direct user interaction.

Applications can register to receive broadcast intents at runtime, which are system-wide intents delivered to each app when certain events happen on the device, such as network changes or the user unlocking the screen. Developers typically register to receive specific broadcasts in the `AndroidManifest.xml` file or dynamically at runtime through code. Then, a broadcast is sent, and the system automatically routes broadcasts to apps that have subscribed to receive that particular type of broadcast. In Android 8 (API level 26), broadcast intent behavior was changed, limiting the implicit intents that applications can register for in the manifest. In most cases, applications that register through the manifest will no longer receive the broadcasts. Now, applications must register context-specific broadcast receivers while the user is actively using the app. Below are examples of relevant system broadcasts that an app can register to achieve or can help achieve persistence

11. See `https://developer.android.com/develop/background-work/background-tasks/broadcasts`

BOOT_COMPLETED: Perhaps the most common and straightforward system event leveraged for persistence is the BOOT_COMPLETED broadcast. It is sent by the Android system once the device has finished booting and all critical services have been initialized. Any application with the RECEIVE_BOOT_COMPLETED permission and a manifest-declared receiver can intercept this broadcast and start a service or activity immediately after every device restart.

```
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />

<receiver android:name=".BootReceiver">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
    </intent-filter>
</receiver>
```

Figure 2.6: Manifest declaration for a BroadcastReceiver that listens for the `BOOT_COMPLETED` event.

PACKAGE_REMOVED: **BrainTest** malware employed a very interesting tactic that tried to gain persistence by creating some sort of a watchdog. It installs two applications - `mcpef.apk` and `brother.apk` - with the same functionality, and these two applications monitor the removal of each other. If one of the applications is deleted, the second application downloads and re-installs the removed one. The alert of uninstallation was done by listening for PACKAGE_REMOVED events[22].

2.5 Abusing system schedulers

Android provides multiple APIs that allow applications to schedule deferred or periodic tasks. These schedulers, originally intended for background processing, efficiency, and battery conservation, are also commonly leveraged by malware to persist over time without continuous execution, often in combination with other components like services and receivers.

Prior to Android 5.0 (Lollipop), most apps used `AlarmManager`, a basic scheduler that could trigger tasks at specific times or inter-

vals—even if the app was not running. However, this led to excessive background wakeups and battery drain, so Android 5.0 introduced the `JobScheduler` API to perform tasks based on system state and constraints (e.g., "only when charging and on Wi-Fi").

Later, `WorkManager`[12] abstracted over both `AlarmManager` and `JobScheduler` to provide a unified scheduling interface that is backward-compatible and survives app restarts.

Malware can exploit these APIs to reactivate itself after being idle, schedule data exfiltration at off-peak times, or trigger payloads after reboot or user inactivity. While these mechanisms do not survive a factory reset or device wipe, they provide a lightweight and low-visibility method of persistence in user space.

`AlarmManager`

`AlarmManager` allows apps to trigger code at exact or repeating time intervals, regardless of the app's running state. Although Android 6.0 introduced Doze mode and battery optimizations that delay alarms, malicious apps still use `AlarmManager` to trigger commands that wake up the app and execute background tasks.

TikTok Pro: This malicious app was discovered in 2020; it impersonated the legitimate TikTok video-sharing platform. According to Zscaler, the app was distributed via phishing SMS and prompted users to install an APK that appeared to offer TikTok functionality but instead delivered a surveillance payload.

TikTok Pro implements `AlarmReceiver` and starts its MainService every time the event occurs. In this case, the alarm was triggered every three minutes. This helped to ensure that the malware would not be terminated by the Android operating system[29].

Timers

In addition to official Android APIs, malware often uses Java-level constructs such as `TimerTask`, `Handler`, or even native loops to create

12. See https://developer.android.com/develop/background-work/background-tasks/persistent/getting-started

custom scheduling logic[13]. These methods are not like Android's system schedulers, which are not subjected to battery constraints and may be harder to detect via static analysis.

TikTok Pro: On top of using the alarms, TikTok Pro used custom Java-level timers, specifically the `java.util.Timer` and `TimerTask` classes, to execute malicious routines like taking care of connection and disconnection to the Command-and-Control (C&C) server at fixed intervals. By pairing this with a `BOOT_COMPLETED` receiver, TikTok Pro ensures that these tasks are rescheduled after every reboot, achieving persistence through a lightweight, user-space mechanism[29].

```
TimerTask task = new TimerTask() {
  public void run() {
    // code to run every second
  }
};

// Schedule the tasks to run every 1 second
this.myTimer.scheduleAtFixedRate(task, 1000L, 1000L);
```

Figure 2.7: Example of scheduling by using the `TimerTask`

Other examples: Another example of malware using timers to schedule communication with C&C server is a trojan named **GPlayed**[30].

2.6 Summary

On the next page is a table showcasing a brief summary of all the methods and their subtechniques discussed in this chapter.

13. See `https://developer.android.com/reference/java/util/Timer`

Table 2.1: Summary of Android malware persistence techniques

Technique	Subtechniques	Root	Reboot / Factory Reset	Activation Phase	Example Malware
Image Tampering	`init.rc`, `install-recovery.sh`, full image replace	Yes	Yes / Yes	Boot time	OldBoot, xHelper, SnapPea
System App Installation	APKs in `/system/app`, `/system/priv-app`	Yes	Yes / Yes	Boot time	BusyGasper, BrainTest, SpyDealer
Binary / Library Injection	Library hijack, binary patch, process injection	Yes	Yes / Partial	Boot / Runtime	Triada, Dvmap, Loki
Update Suppression	OTA block, updater removal, auto-update disabled	Yes	Yes / No	Runtime	Pegasus
Accessibility Services	UI event monitor, input spoofing, overlay inject	No	Yes / No	Runtime	SharkBot, Anubis, Cerberus
Broadcast Receivers	BOOT_COMPLETED, PACKAGE_REMOVED, CONNECTIVITY_CHANGE	No	Yes / No	Boot / Runtime	BrainTest, Shedun, Gooligan
System Schedulers	`AlarmManager`, `JobScheduler`, `TimerTask`	No	Yes / No	Runtime	Joker, GPlayed, TikTok Pro
Service Abuse	Foreground/background with restart logic	No	Yes / No	Runtime	Mandrake, BOULDSPY
UI/UX Manipulation	Fake shutdown, icon hiding, spoofed dialogs	No	No / No	Runtime	Predator, Alien

3 Case Study of a Persistent Malware: *Triada*

Triada is one of the most technically advanced Android malware families ever identified. First reported in 2016, Triada evolved from an app-layer rooting Trojan into a modular, firmware-resident backdoor capable of deep system integration and long-term persistence. It is particularly notable for being the first publicly documented Android malware to inject malicious code into `libandroid_runtime.so`, thereby hooking the Zygote process, a fundamental system component that spawns all app processes[31].

Over time, Triada adopted a multi-layered persistence strategy combining system app installation, shared library injection, and remote module loading. By 2017, Triada had been integrated into the firmware images of commercial Android devices, reflecting a shift toward supply chain–level persistence[32, 33].

3.1 Initial Infection and Delivery

Triada's initial variants were distributed via repackaged apps in third-party marketplaces. These apps downloaded a native library and used known exploits (e.g., CVE-2013-6282, CVE-2014-3153) to gain root access. Once privileged, the malware modified system directories and installed components with elevated permissions.

In 2017, Google confirmed that Triada had transitioned to a supply chain threat, appearing pre-installed in low-cost Android smartphones during the firmware assembly process[32]. The malware was embedded by third-party firmware providers and injected into system partitions as part of custom ROM builds. Often disguised as innocuous apps like `com.android.camera.update`, these components operated with system-level privileges from first boot.

3.2 Persistence Techniques

Triada's architecture showcases layered persistence, combining system-level installation with user-space injection and runtime control.

System Application Installation 2.1 Triada placed one or more of its APKs in `/system/priv-app/`, granting them default access to privileged permissions. These apps were automatically launched at boot, often invisible to the user. As system apps, they could bypass runtime permission prompts and avoid uninstallation via the standard UI.

Library Injection via `libandroid_runtime.so` 2.1 In its most sophisticated phase, Triada modified `libandroid_runtime.so`, a core system library loaded by Zygote, Android's process initiator[31]. This allowed Triada to inject its logic into every new application process, giving it global runtime visibility and control. Unlike typical process injection, this method enabled persistent code execution across all apps on the device without touching their binaries directly.

Zygote Process Hooking Because of its library injection, Triada effectively hooks into the Zygote process itself. Since Zygote spawns all user apps, hooking it allowed Triada to ensure that every new app process inherited its malicious logic. This model is functionally similar to system-wide runtime instrumentation and made Triada exceptionally difficult to detect using per-app analysis.

Modular Loader Architecture Triada operated as a plugin loader: its system component connected to a command-and-control (C2) server, received instructions, and downloaded plugin modules to execute various payloads. These included ad fraud engines, SMS interceptors, spyware tools, and proxy modules. This modularity made Triada highly adaptable and updatable in the field [34].

3.3 Capabilities and Behavior

Triada's primary objectives included monetization and surveillance, achieved through its core capabilities:

- Perform ad fraud by displaying and clicking hidden ads.
- Subscribe users to premium SMS services.
- Intercept user SMS messages and exfiltrate contact data.

- Download and install additional APKs without consent.
- Proxy network traffic through the device for anonymity or further abuse.

In later variants, Triada's modular framework evolved into a firmware-level "malware platform," effectively embedding a persistent loader within legitimate-looking system apps shipped with the device[34].

3.4 Attribution and Motivation

According to Google's Android security team, Triada's firmware-level integration was not performed by device manufacturers directly, but rather by third-party firmware integrators or vendors who incorporated a malicious SDK into legitimate system apps[32]. The SDK included Triada's plugin loader, granting attackers full post-deployment access to the device.

Krebs on Security independently confirmed that the firmware supply chain had been compromised, with some vendors unknowingly shipping infected devices to global markets[33].

This evolution highlights how persistence can migrate from app-based attacks to pre-installed malware with privileged access from first boot.

Table 3.1: Timeline of Triada's Evolution

Year	Milestone
2016	First variant discovered; used root exploits and injected into `libandroid_runtime.so`, enabling control over Zygote and all spawned app processes.
2017	Triada began appearing pre-installed on commercial Android devices. This marked the beginning of its shift to supply chain compromise.
2019	Google and independent researchers confirmed that Triada was injected into firmware by third-party SDK vendors during production.
2023–2025	New variants observed in counterfeit or budget smartphones. Triada evolves into a firmware-resident plugin loader and modular malware platform.

4 Implementation

While the primary focus of this thesis is the theoretical and technical classification of persistence mechanisms in Android malware, a practical implementation was developed to demonstrate how selected techniques can be applied in a controlled, real-world environment. The goal was to simulate a malware-like Android application on a physical device, capable of maintaining persistence across device reboots and factory resets by combining user-space and system-level techniques. This chapter outlines the selected technique, the tools and environment used for implementation, the steps taken during development, and the limitations encountered during testing.

4.1 Implemented Techniques

The implementation focused on two complementary persistence techniques:

- System application installation into `/system/priv-app/`, as described in chapter 2.1: Installing as System Application.
- Use of a `BOOT_COMPLETED` `BroadcastReceiver`, as described in chapter 2.4: Abusing receivers.

These methods were chosen for their prevalence in real-world Android malware families and their ability to demonstrate layered persistence.

4.2 Tools and Device Setup

The implementation was tailored to a physical **Redmi Note 8T** device. It may or may not work for other rooted devices.

The setup included:

- **Bootloader unlock:** Using Xiaomi's official tool[35], following community instructions on XDA Developers[36].
- **Custom recovery:** OrangeFox Recovery[37].
- **Custom ROM:** crDroid 10.7 (Android 14) for ginkgo[38].

- **Root access:** Magisk with `libsu` for root shell execution[39].
- **Development tools:** Android Studio[40], and Android platform tools on Arch Linux[41].

Unlocking the bootloader: The Redmi Note 8T device was chosen for implementation since it had been unlocked previously, which eliminated delays associated with Xiaomi's unlock waiting period. However, the unlocking took multiple attempts because the official app failed multiple times, highlighting the practical challenges of tampering with protected partitions, even in a legitimate research context.

Flashing Custom Recovery and ROM: After the bootloader was unlocked, it was time to try and flash custom recovery. At first, the TWRP was chosen and reflashed multiple times, however, always resulting in issues and failures. This was the reason for flashing a less popular and less known recovery named OrangeFox. Thankfully, no issues were present. With a functional recovery, the rest was pretty simple. The crDroid ROM was chosen since it had an available build based on Android 14 for the Redmi Note 8T device.

Gaining root access: Gaining root access via Magisk was accomplished by following the instructions via patching the `boot.img`.

4.3 Development

SDK Configuration

The application was built with the following Android SDK parameters:

- **Minimum SDK version (`minSdk`): 24**
  Corresponding to **Android 7.0 (Nougat)**, this was selected because it supports key components such as `BOOT_COMPLETED` broadcast receivers and modern root access frameworks (e.g., Magisk and `libsu`). It represents the baseline version for many

malware techniques that rely on background services and broadcast triggers without the constraints introduced in later Android versions.

- **Target SDK version (`targetSdk`): 35**
  Corresponding to **Android 14**, this setting ensures compatibility with the most recent API behavior and security restrictions. It allows the application to demonstrate how persistence mechanisms function under current platform constraints, such as background execution limits and stricter broadcast registration policies introduced in Android 8.0+.

This configuration reflects a practical balance between backward compatibility—allowing implementation of persistence techniques still relevant to many Android variants in circulation—and forward compatibility with modern Android security architecture.

Core mechanism

LoggingActivity.java This serves as the launcher activity. It initializes the root shell using `libsu` and starts the `LoggingService`. It ensures that the application has root privileges before attempting any system-level operations.

LoggingService.java This background service performs two functions:

- Installs the APK into the `/system/priv-app/` directory if root access is available.
- Starts a logging thread that simulates background activity.

It uses `START_STICKY` to persist across kills and attempts to remount the system partition, copy the APK, set permissions, and then reboot.

BootReceiver.java This component listens for the `BOOT_COMPLETED` broadcast and starts the service after reboot. This simulates malware using boot-time activation for reinitialization.

4.4 Results

After installation, the application :

- Checks if the current path is in `/system`
  - If not, it copies itself into the `/system` partition and initializes reboot to register it and run it as a system app immediately. In a usual setting, this would be triggered by turning off or rebooting by the user. In this application, the rebooting was hardcoded to showcase the functionality.
  - In case it already runs as a system app, it starts a logging service in the background, simulating malicious behaviour.
- Since it gains privileges as a system application, the user will not be able to uninstall it.
- Manages to survive reboots.
- Manages to survive factory resets with minor issues described in the next section 4.5.

4.5 Issues and limitations

This proof-of-concept is intentionally limited:

- It does not implement malicious behavior.
- It does not contain any obfuscation or any other method trying to hide the application and its function.
- It assumes an already rooted device via exploit or other technique available to an attacker.
- Even as a system application, the app can be killed by a user and will be stopped after a few minutes of inactivity by the system to prevent battery draining. This results in the background service being killed. This behavior can be easily subverted by

implementing other additional methods mentioned in this thesis. The main goal of this PoC is, however, the installation on a protected partition and automatic startup on boot.

During the testing phase, we encountered persistent issues that we could not resolve:

- Performing a factory reset resulted in corruption of the `/data` partition, leading the device to boot into OrangeFox recovery.
- After initializing factory reset and formatting the data partition in recovery, the ROM booted normally. After rebooting, it started an initial setup of the device, and the app remained installed and functioning with root access preserved.
- This behavior is likely due to Magisk preserving its modules or patched boot images, even after a reset.

Nonetheless, it effectively demonstrates how basic Android malware can layer multiple persistence techniques to maintain control over a device.

Conclusion

Persistence remains one of the most strategically important objectives for Android malware. As this thesis has demonstrated, persistence can be achieved through a diverse set of techniques that target different layers of the Android operating system—from low-level firmware modifications and system partition tampering, to user-space abuses of services, receivers, and UI deception. These techniques are not mutually exclusive; in fact, modern malware frequently combines multiple mechanisms to increase resilience, delay detection, and outlast user countermeasures such as reboots or factory resets.

Through the structured analysis of persistence strategies and their classification by system component, this work contributes to a clearer technical understanding of how Android malware maintains its presence. Furthermore, the in-depth case study on Triada illustrates how these techniques evolve in practice. Triada's use of system app installation, runtime hooking, and supply chain compromise exemplifies how malware can exploit both software design and the economics of device manufacturing to establish a long-term foothold.

The practical implementation developed for this thesis simulates a simplified version of real-world malware behavior. It demonstrates how, on a rooted device with an unlocked bootloader, an application can install itself into the system partition and use broadcast receivers to persist across reboots. While such capabilities may seem extreme, they remain relevant for researchers, forensic analysts, and security professionals who often operate in environments where devices are already compromised or where malware authors control parts of the firmware deployment chain.

Importantly, this work also highlights the limitations of existing defense mechanisms. As seen in the discussion of Verified Boot and Android Verified Boot 2.0 (AVB), even strong integrity enforcement systems can be bypassed when the bootloader is unlocked or when malicious components are injected at the supply chain level. Although these protections raise the bar for adversaries, they do not eliminate the possibility of deeply embedded persistence without user consent.

Finally, it is important to note that persistence techniques cannot be examined in isolation. Their value to adversaries lies in their ability

to support broader malware objectives, such as surveillance, data theft, and monetization. Therefore, this thesis naturally leads to the next major topic in Android security research: malware detection and protection. Effective defense strategies must not only block known persistence techniques but must also anticipate their evolution and integration with other malicious functionalities.

By understanding how persistence works, defenders can better detect the conditions that make it possible—and ultimately disrupt the long-term strategies of Android malware operators.

Bibliography

1. GOOGLE. *The First Android-powered Phone*. 2008-09-23. Available also from: https://web.archive.org/web/20250428110553/https://googleblog.blogspot.com/2008/09/first-android-powered-phone.html.
2. STATISTA. *Market Share of Mobile Operating Systems Worldwide 2009-2025, by Quarter* [online]. 2025-03-11. [visited on 2025-05-18]. Available from: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.
3. HAN, Qian; MANDUJANO, Salvador; PORST, Sebastian; SUBRAHMANIAN, V.S.; TETALI, Sai Deep. *The Android Malware Handbook: Detection and Analysis by Human and Machine*. No Starch Press, 2023. ISBN 978-1-7185-0331-1.
4. LEVIN, Jonathan. *Android Internals: A Confectioner's Cookbook: Volume 1 : the Power Users's View*. Technologeeks.com, 2015. ISBN 978-0-9910555-2-4.
5. ANDROID OPEN SOURCE PROJECT DEVELOPERS. *Android OS Source Documentation: Core Topics - Architecture - Partitions* [online]. 2025-04-04. [visited on 2025-05-12]. Available from: https://source.android.com/docs/core/architecture/partitions.
6. BUCHKA, Nikita. Taking Root. *Securelist*. 2015. Available also from: https://web.archive.org/web/20250403214350/https://securelist.com/taking-root/71981/.
7. ZACUTO, Jeff. *Adware or APT – SnapPea Downloader – an Android Malware That Implements 12 Different Exploits* [online]. 2015-07-10. [visited on 2025-05-19]. Available from: https://web.archive.org/web/20240429170320/https://blog.checkpoint.com/research/adware-or-apt-snappea-downloader-an-android-malware-that-implements-12-different-exploits/.

8. HU, Wenjun; XIAO, Claud; XU, Zhi. *Rootnik Android Trojan Abuses Commercial Rooting Tool and Steals Private Information* [online]. 2015-12-04. [visited on 2025-05-19]. Available from: https://web.archive.org/web/20250421035356/https://unit42.paloaltonetworks.com/rootnik-android-trojan-abuses-commercial-rooting-tool-and-steals-private-information/.

9. LU, Kai. *Deep Analysis of Android Rootnik Malware Using Advanced Anti-Debug and Anti-Hook, Part II: Analysis of The Scope of Java* [online]. 2017-01-26. [visited on 2025-05-19]. Available from: https://web.archive.org/web/20250505032023/https://www.fortinet.com/blog/threat-research/deep-analysis-of-android-rootnik-malware-using-advanced-anti-debug-and-anti-hook-part-ii-analysis-of-the-scope-of-java.

10. TRUSTLOOK. *A Trojan With Hidden Malicious Code Steals User's Messenger App Information* [online]. 2018-04-02. [visited on 2025-05-19]. Available from: https://web.archive.org/web/20180723182741/https://blog.trustlook.com/2018/04/02/a-trojan-with-hidden-malicious-code-steals-users-messenger-app-information/.

11. GOLOVIN, Igor. Unkillable xHelper and a Trojan Matryoshka. *Securelist*. 2020. Available also from: https://web.archive.org/web/20250323004450/https://securelist.com/unkillable-xhelper-and-a-trojan-matryoshka/96487/.

12. DOCTOR WEB. *Android.Oldboot.1: Dr.Web Malware Description Library* [online]. 2015-07-15. [visited on 2025-05-20]. Available from: https://web.archive.org/web/20240803123542/https://vms.drweb.com/virus/?i=4627207.

13. LOOKOUT. *Pegasus for Android: Technical Analysis and Findings of Chrysaor* [online]. 2017-04. [visited on 2025-05-20]. Available from: https://web.archive.org/web/20250513105833/https://info.lookout.com/rs/051-ESQ-475/images/lookout-pegasus-android-technical-analysis.pdf.

14. CANNINGS, Rich; WOLOZ, Jason; MEHTA, Neel; BODZAK, Ken; CHANG, Wentao; RUTHVEN, Megan. An Investigation of Chrysaor Malware on Android. *Android Developers Blog* [online]. 2017 [visited on 2025-05-20]. Available from: https://

web.archive.org/web/20250504083529/https://android-developers.googleblog.com/2017/04/an-investigation-of-chrysaor-malware-on.html.

15. UNUCHEK, Roman. Dvmap: The First Android Malware With Code Injection. *Securelist*. 2017. Available also from: https://web.archive.org/web/20250430112631/https://securelist.com/dvmap-the-first-android-malware-with-code-injection/78648/.

16. SIEWIERSKI, Lukasz; ANDROID SECURITY & PRIVACY TEAM. *PHA Family Highlights: Zen and Its Cousins* [online]. 2019-01-11. [visited on 2025-05-20]. Available from: https://web.archive.org/web/20250429224524/https://security.googleblog.com/2019/01/pha-family-highlights-zen-and-its.html.

17. ERGUVEN, Yusa. *Process Execution Methods in Android* [online]. 2017-08-28. [visited on 2025-05-20]. Available from: https://web.archive.org/web/20200917210258/https://yusa.github.io/android/malware/cybersecurity/2017/08/28/earlier-post.html.

18. DOCTOR WEB. *Android.Loki.3: Dr.Web Malware Description Library* [online]. 2016-02-05. [visited on 2025-05-20]. Available from: https://vms.drweb.com/virus/?i=7954502&lng=en.

19. CHECK POINT MOBILE RESEARCH TEAM. *From HummingBad to Worse* [online]. Check Point, 2016 [visited on 2025-05-21]. Available from: https://web.archive.org/web/20250430052840/https://blog.checkpoint.com/wp-content/uploads/2016/07/HummingBad-Research-report_FINAL-62916.pdf.

20. GOODIN, Dan. 10 million Android phones infected by all-powerful auto-rooting apps. *Ars Technica*. 2016. Available also from: https://web.archive.org/web/20250317215413/https://arstechnica.com/information-technology/2016/07/virulent-auto-rooting-malware-takes-control-of-10-million-android-devices/.

21. FIRSH, Alexey. BusyGasper: the unfriendly spy. *Securelist*. 2018. Available also from: https://web.archive.org/web/20250427053713/https://securelist.com/busygasper-the-unfriendly-spy/87627/.

22. POLKOVNICHENKO, Andrey; BOXINER, Alon. *BrainTest: a New Level of Sophistication in Mobile Malware* [online]. 2015-09-21. [visited on 2025-05-22]. Available from: https://web.archive.org/web/20230423062911/https://blog.checkpoint.com/research/braintest-a-new-level-of-sophistication-in-mobile-malware/.

23. HU, Wenjun; ZHENG, Cong; XU, Zhi. *SpyDealer: Android Trojan Spying on More Than 40 Apps* [online]. 2017-07-06. [visited on 2025-05-22]. Available from: https://web.archive.org/web/20250430114845/https://unit42.paloaltonetworks.com/unit42-spydealer-android-trojan-spying-40-apps/.

24. LOOKOUT THREAT INTELLIGENCE TEAM. *Mobile APT Surveillance Campaigns Targeting Uyghurs: A collection of long-running Android tooling connected to a Chinese mAPT actor* [online]. Lookout, 2020-06 [visited on 2025-05-22]. Available from: https://web.archive.org/web/20250426104849/https://www.lookout.com/documents/threat-reports/us/lookout-uyghur-malware-tr-us.pdf.

25. SCHMITTLE, Kyle; ISLAMOGLU, Alemdar; SHUNK, Paul; ALBRECHT, Justin. *BouldSpy: Android Spyware Tied to Iranian Police Targets Minorities* [online]. 2023-04-27. [visited on 2025-05-22]. Available from: https://web.archive.org/web/20250228013412/https://www.lookout.com/threat-intelligence/article/iranian-spyware-bouldspy.

26. GEVERS, Rickey; TIVADAR, Marius; BLEOTU, Rareș; BARBATEI, Alin Mihai; BALÁZS, Bíró; COBLIȘ, Claudiu. *Uprooting Mandrake: The Story of an Advanced Android Spyware Framework That Went Undetected for 4 Years* [online]. Bitdefender, 2020-05-14 [visited on 2025-05-22]. Available from: https://web.archive.org/web/20250427011123/https://www.bitdefender.com/files/News/CaseStudies/study/329/Bitdefender-PR-Whitepaper-Mandrake-creat4464-en-EN-interactive.pdf.

27. SHISHKOVA, Tatyana; GOLOVIN, Igor. Mandrake spyware sneaks onto Google Play again, flying under the radar for two years. *Securelist*. 2024. Available also from: https://web.archive.org/

web/20250516033742/https://securelist.com/mandrake-apps-return-to-google-play/113147/.

28. CISCO TALOS. *Mercenary Mayhem: A Technical Analysis of Intellexa's PREDATOR Spyware* [online]. 2023-05-25. [visited on 2025-05-22]. Available from: https://web.archive.org/web/20250315024904/https://blog.talosintelligence.com/mercenary-intellexa-predator/.

29. DESAI, Shivang. *TikTok Spyware* [online]. 2020-09-08. [visited on 2025-05-22]. Available from: https://web.archive.org/web/20250210033028/https://www.zscaler.com/blogs/security-research/tiktok-spyware.

30. VENTURA, Vitor. *GPlayed Trojan: .Net Playing With Google Market* [online]. 2018-10-11. [visited on 2025-05-22]. Available from: https://web.archive.org/web/20250506015109/https://blog.talosintelligence.com/gplayedtrojan/.

31. KASPERSKY. *Attack on Zygote: A New Twist in the Evolution of Mobile Threats* [online]. 2016. [visited on 2025-05-22]. Available from: https://securelist.com/attack-on-zygote-a-new-twist-in-the-evolution-of-mobile-threats/74032/.

32. BLOG, Google Security. *PHA Family Highlights: Triada* [online]. 2019. [visited on 2025-05-22]. Available from: https://security.googleblog.com/2019/06/pha-family-highlights-triada.html.

33. KREBS, Brian. *Tracing the Supply Chain Attack on Android* [online]. 2019. [visited on 2025-05-22]. Available from: https://krebsonsecurity.com/2019/06/tracing-the-supply-chain-attack-on-android-2/.

34. KASPERSKY. *Triada Trojan Modules: Analysis of the Persistence Platform* [online]. 2025. [visited on 2025-05-22]. Available from: https://securelist.com/triada-trojan-modules-analysis/116380/.

35. *Mi Unlock Tool*. 2025. Available also from: https://en.miui.com/unlock/download_en.html.

36. *Guide: Unlocking, TWRP, and Rooting Redmi Note 8/8T*. 2025. Available also from: https://xdaforums.com/t/guide-all-about-unlocking-bootloader-twrp-and-rooting-redmi-note-8-8t.4031831/.

37. *OrangeFox Recovery Project*. 2025. Available also from: https://wiki.orangefox.tech/.

38. *crDroid 10.7 for ginkgo*. 2025. Available also from: https://crdroid.net/ginkgo/10.

39. TOPJOHNWU. *Magisk Installation Guide*. 2025. Available also from: https://topjohnwu.github.io/Magisk/install.html.

40. *Android Studio*. 2025. Available also from: https://developer.android.com/studio.

41. *android-tools - Arch Linux Package*. 2025. Available also from: https://archlinux.org/packages/extra/x86_64/android-tools/.

A An appendix

A complete archive of the implementation project is included as supplementary material in the form of a compressed `.zip` file. This archive contains:

- Full Android Studio project with source code and Gradle build scripts.
- Configuration files, including `libs.versions.toml`, `build.gradle.kts`, and `AndroidManifest.xml`.
- A `README.md` file describing the project structure, dependencies, and installation procedure.