Android, Notify Me When It Is Time To Go Phishing

Antonio Ruggia, Andrea Possemato, Alessio Merlo, Dario Nisi, Simone Aonzo

▶ **To cite this version:**

Antonio Ruggia, Andrea Possemato, Alessio Merlo, Dario Nisi, Simone Aonzo. Android, Notify Me When It Is Time To Go Phishing. EUROS&P 2023, 8th IEEE European Symposium on Security and Privacy, IEEE, Jul 2023, Delft, Netherlands. hal-04017843

HAL Id: hal-04017843
https://hal.science/hal-04017843

Submitted on 7 Mar 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Android, Notify Me When It Is Time To Go Phishing

Antonio Ruggia*, Andrea Possemato†, Alessio Merlo*, Dario Nisi†, Simone Aonzo†
* *University of Genoa, Italy*
† *EURECOM, France*

***Abstract*—A mobile banking app just started up, and the notification "App updated, click here to restart" appears. The graphic theme is the same as the bank. Can we trust it? What if we cannot even trust that tapping an app actually loads the original one? More generally, what if Android notifies an attacker when her victim has just launched the target app of her phishing campaign so that she could cast the hook at the perfect moment?**

In this paper, we abuse inotify APIs, a mechanism for monitoring file system events, to mount a state inference-based phishing attack from a malicious app installed on the victim's smartphone. We also verified the novelty of our work analyzing 10,000 recent Android malware, and although we found some cases where malware uses inotify for their petty purposes, our attack seems to be publicly unknown.

However, since Android constantly evolves year after year, we studied its feasibility over different Android versions and attacker's capabilities. By analyzing $4{,}863$ of the most popular apps, the most disconcerting finding is that if the attacker knows the installation path of the target app, *all* Android apps are vulnerable, regardless of the system version. Getting the installation path of an app is a capability that is only protected by a normal permission, and to make matters worse, there are workarounds to get it even without such permission.

Even if this capability is denied, we propose different attack models under which this attack is still possible; however, at the end of our work, we provide the remediation to eradicate once and for all these attacks. Through this work, we reported three vulnerabilities to Google. Two were acknowledged as bugs of moderate severity, while the last one was already known but not public.

1. Introduction

File system event monitoring is essential for many programs, from file managers to security tools. Starting from version 2.6.13, the Linux kernel includes *inotify* [31]—a component that allows a userspace program to monitor and react to file system events, such as file accesses, deletions, and modifications. The *inotify* [31] APIs allow targeting either singular files or entire directories and filtering only specific events. When one of such events occurs on the monitored files, the system notifies the app, which, in turn, can execute code in response to the event.

Android has fully supported inotify since its early versions (API Level 1 [27]). Apps can *watch* file system *events* – employing the inotify terminology – from native code (C/C++) through the standard inotify APIs, and Java/Kotlin, using the `FileObserver` class.

While providing powerful primitives to implement new features, the inotify infrastructure has also played a role in Man-in-the-Disk (MitD) attacks [32]. These attacks are possible when the program's logic depends on the content of a file that an attacker can access and modify. Inotify proved perfect for this task. By watching for a specific event on the target file, the attacker can timely and opportunistically replace its legitimate content with malicious data. The consequences may be more or less severe depending on the tampered data.

In 2016, Ahn et al. [3] showed that particular file system events' occurrences allow detecting a specific UI screen transition. Once again, inotify made it possible to monitor specific file system patterns, this time for perfectly timing a phishing attack by starting an activity that resembles the original one. While inspiring, this work relies on capabilities that attackers can no longer obtain under reasonable threat models (e.g., Android does not allow third-party apps to list running apps and their PIDs).

Given its history of misuse for malicious purposes, this work assesses whether the inotify subsystem can still play a role in *state inference attacks,* despite the tight security models the Android developers put in place. This attack class aims to determine the state of another app and pave the way to a large class of attacks, including *phishing*. In particular, this work focuses on determining when an app starts. At the same time, it is the most valuable for an attacker and the most practical to analyze automatically. From an attacker's perspective, determining when the user willingly launches an app maximizes the chances to lure it into providing accounts credential to a phony UI interface. It is not by chance that banking malware often monitors apps' startup for phishing purposes [30], [54]. By choosing to study the app's startup and not any generic UI transitions, we do not need heavy human intervention to select and trigger valuable ones (e.g., login screens). The concept behind our approach is straightforward: If an app always generates a *unique* file system event at startup (among other apps installed on the device) and an attacker can monitor such an event with inotify, she can easily infer when the user has just opened the app.

We manually tested the feasibility of such an attack by creating a vulnerable target app that reads a world-readable text file when its main activity starts and a malicious app that monitors these events with inotify. When the vulnerable app was executed, the malicious app was "notified" and could push a new phishing activity. Picture the scene: the user taps on the target app icon, and the new fake spoofed activity that looks like the original one is displayed on the screen; this leaves no trace to the victim, who has no way of noticing the deception and distinguishing between the original and the malicious.

However, this precise attack is limited by a privileged Android permission (`SYSTEM_ALERT_WINDOW`) that

requires a slightly more complicated procedure to get it from the victim. Although this does not stop modern malware authors from using it (according to our measurements, 57% of malware requires it, while this percentage plummets to 9% for benign apps), the attack can still be performed by popping up a fake notification that seems to come from the target app.

For this reason, in this paper, we investigate the conditions surrounding the feasibility of this attack, trying to answer the following research questions:

- **RQ1** Are malware authors already exploiting inotify?
- **RQ2** What capabilities (e.g., permissions or information) must an attacker have to carry out this attack?
- **RQ3** To what extent does the Android version affect this attack and the attacker's capabilities?
- **RQ4** Are the attacker's capabilities realistic to carry out this state inference attack and mount a phishing attack?
- **RQ5** How is the security posture of other vendors concerning this attack?
- **RQ6** Which mitigations can stop this attack for good?

First, we investigated the usage of inotify among recent Android malware to assess the novelty of our attack. Our results on a set of over $10{,}000$ malware in 2021 show that they abuse inotify only for implementing anti-debugging tricks or executing code when they are uninstalled. In the other cases, we detected the use of inotify just from analytics libraries. Therefore, to the best of our knowledge, no existing Android malware implements the attack pattern we discuss in this paper.

Then, we investigated the conditions for carrying out this attack. Given that the attacker's capabilities depend on which files the apps use when started and if she has the permission to add an inotify watch on such files, we developed *inoTool*, a tool to dynamically analyze Android apps and monitor their interactions with the file system.

With *inoTool*, we analyzed a heterogeneous dataset of $4{,}863$ benign Android app, created by collecting the most downloaded free apps over the 50 categories in the Google Play Store. We found that an attacker can generate a kind of "signature" that uniquely identifies which files her target app opens at startup, enabling her to launch a phishing attack by monitoring one of such files. This attack is feasible on a large scale, and during our investigation, we uncovered some design issues that improve the likelihood of success. For example, we discovered an information disclosure vulnerability through which one can get a partial list of the apps installed on a device, despite the recent changes in Android strongly restricting package visibility, namely, querying for information about the other apps.

We identified three attack models from the analysis of the results and the various mechanisms involved. As the main finding of our work, the most potent model (in which an attacker is assumed to know the installation path of its target app), **all** Android apps are vulnerable, regardless of the Android version. Moreover, the only capability needed to implement this attack is protected by a single permission automatically granted at installation time. Although Google is aware of the risks deriving from this permission and performs capillary checks on its Play Store, we will also show that the current security model of Android presents several loopholes that make obtaining this permission unnecessary.

The second attack model assumes that an attacker can no longer obtain the installation paths of the apps installed on the device. Under these circumstances, we show that an attacker still manages to implement the attack on a limited number of apps. The third and last model refines the strategy further and manages, on average, to target half of the apps installed on a typical device.

Finally, we show how different remedies defeat these different types of attackers. However, in some cases, the specific behavior of the target app, in combination with the apps installed on the device, still leaves a small window for successfully carrying out the attack.

In the spirit of open science, we release all the source code we developed during our study [42]. On the other hand, we cannot share benign apps or malware due to legal restrictions, but we share their hashes.

2. Background

This section provides the necessary technical background for the rest of the paper. We start by presenting the relevant details of the Android operating system, and then we continue by introducing the *inotify* subsystem of the Linux kernel.

2.1. Android Internals

Android Apps' Anatomy and Installation. Apps are the cornerstone of the Android ecosystem and represent the primary vehicle for developers to provide new features to users. Java and Kotlin are the reference languages for app development; then, they are semi-compiled in *Dalvik bytecode* (DEX) files. While very versatile, these languages and the Dalvik technology are generally not optimal for performance-critical tasks or low-level interactions with the device hardware. To deal with this shortcoming, the Android Software Development Kit (SDK) also allows writing portions of an app in C/C++, which are compiled into native libraries. During the app-building process, the SDK packs the Dalvik bytecode and the native code into an Android Package (APK) file that the developer can then distribute, e.g., through app marketplaces. Beyond the code, an APK file contains resources required for the app execution (e.g., images to render the user interface) and a *Manifest* file. The Manifest provides valuable information about the app, such as its unique package name and required permissions.

Permissions allows Android to protect sensitive data (e.g., system state and a user's contact information) and prevents potentially dangerous actions (e.g., connecting to a Bluetooth device or recording audio). Android permissions are ranked into different protection levels [22] according to the sensitivity of the resource they protect. These levels effectively implement a layered clearance mechanism. While the system automatically grants *normal* permissions (i.e., those that present minimal risk to the user's privacy) at installation time, *dangerous* (or *runtime*) permissions need explicit runtime authorization by the user. Each time an app tries to access a resource or perform an action protected by dangerous permissions, the user is prompted with a UI message detailing the type of resource/action the app attempted to access and asking whether to grant or deny it. The highest protection level is the *privileged* one. Permissions with this level have

an impact on the overall security posture of the device. The user has to manually grant them through a multistep procedure in the System Settings app, during which the system warns her about the risks of this permission.

At installation time, Android creates a dedicated directory for the app in `/data/app/` in which the system copies the original APK, renaming it as "`base.apk`." The name of this directory – from now on *installation path* – consists of the app package name and, since Android 8.0 [18], a random string. The random component in the directory name prevents a malicious app from trivially determining another app installation path and accessing its private files. According to the Android documentation, the only supported way to obtain a third-party app installation path is through the `getPackageInfo` API of the Package Manager, which is subject to the permission mechanism. For example, suppose to install an app whose package name is `com.example` on a device with an `x86` CPU. A legit installation path could be `/data/app/com.example-Bd6GIb47XTzpL16==/`. In this location, the system copies the app's APK, renaming it `base.apk` file, and creates a subdirectory named `lib/x86/` that contains all the x86 libraries (`.so`).

Mandatory Access Control & SELinux. Security Enhanced Linux (SELinux) is a mandatory access control (MAC) system for the Linux operating system, and it was introduced in Android 4.3. A MAC system differs from Linux's discretionary access control (DAC) system. Indeed, with a DAC system, an owner of a particular resource controls access permissions associated with it, while on a MAC there is a central authority for a decision on all access attempts. In Android, DAC and MAC coexist: a user can interact with a resource if she has enough permissions (DAC) and the security policy allows it (MAC).

In detail, SELinux adds opaque pointers to potentially sensitive kernel objects (e.g., files, network interfaces) and mediates access to such objects by placing *hooks* functions to determine if an action (known as *permission*, such as `read` or `write`) should be allowed. The decision is made according to the *policies*, which are a set of *rules* that guide SELinux in the choice based on the *labels* of the resources. In fact, every object (e.g., files, directories, processes, etc.) has associated a *security context* (or *label*) that specify its security-related information, i.e., SELinux user, role, type, and sensitivity. In particular, Android relies on the Type Enforcement (TE) component of SELinux, in which the policies are enforced based on the type of subjects and objects. By default, SELinux denies all requests except the ones that respect the current policy. In addition, on Android, a third-party app is assigned to the context with the type `untrusted_app`.

A SELinux policy rule comes in the form `allow sourcetarget:classpermissions`. This rule allows a subject (of type *source*) to access the object (of type *target*). The *class* field specifies the kind of the object (e.g., file, socket, etc.), while the set of involved operations is specified by the *permissions* field. For instance, the `allowuntrusted_appapp_data_file:file{read write}` policy specifies that untrusted apps are allowed to read and write files labeled "app_data_file". Finally, Android provides a set of macros to handle the most common cases to simplify the rule definition. For instance, it defines the permission `rw_file_perms`, which automatically includes all the operations needed to read and write a file.

2.2. Inotify in Linux and Android

Many user apps in modern computer systems need access to the file system beyond simply reading/writing files. Take a remote directory synchronization utility as an example. This utility needs to monitor the target directory to detect file creations, deletions, and modifications to operate automatically. While possible, implementing this type of monitoring by employing traditional file system operations (e.g., listing the directory every second by relying on a busy-waiting model) is not efficient. For this reason, modern operating systems provide dedicated APIs for implementing low-overhead file system monitoring.

One such API is the *inotify* subsystem of the Linux kernel, which provides an event-driven interface to intercept file system events. The canonical *inotify* workflow starts with a program creating an inotify instance through the `inotify_init` syscall and specifying what type of events to monitor through the `inotify_add_watch` syscall. The inotify instance can then be queried with the `read` syscall that blocks the program execution until one of the monitored events occurs.

The Android operating system wraps this workflow in the `FileObserver` abstract class, which programmers can use as a base class to implement file system monitoring. This class constructor takes the file paths to monitor as input, while two more methods start and stop the file system monitoring. Finally, the app can specify the event handling logic by implementing the `onEvent` abstract method that the framework invokes each time an event involving the monitored paths occurs.

3. Related Work

In 2011, Felt et al. [13] first discussed the risk of phishing attacks on mobile platforms and provided a set of techniques to carry out such attacks. As the authors pointed out, an essential ingredient of a successful phishing attack is to intercept the execution of a legitimate app promptly and to overtake the UI to present the user with a faithful copy of the interface of the intended app. When facing a seemingly familiar interface, the user is lured into providing sensitive information, such as credentials. While back in 2011, it was relatively easy to determine when an app started (any app could list all the processes running on the system and implement a *poll-like* wait behavior), things became much harder once phishing attacks became prevalent. As a result, process listing was soon forbidden, forcing attackers to find new ways to detect when the user launched an app. These new techniques are commonly known in the literature as *State Inference Attacks*. Researchers showed that real-world malware [30], [35], [54] employ state inference as the first stage in their attacks, either for *phishing* or to monitor and profile the user. Several research works have studied and documented techniques to reliably determine when an Android app is being executed [6], [11], [14], [16], [38], [40], [46], [47]. Possemato et al. [38] grouped all existing vulnerabilities into two macro-categories:

I) File system layer. The `proc` file system (i.e., `procfs`) is a pseudo-file system that provides an interface to kernel data structures. The files in the `procfs` contain sensitive information, such as the program's name currently executed. Several works [9], [11], [46] discussed different techniques that allow a malicious user to build state inference attacks leveraging the `procfs` information. In addition, Spreitzer et al. [46] proposed a fully automated technique to assess the `procfs` information leaks.

II) Android System Services layer. Android apps use the APIs the System Services of the Android framework exposes to access Android-specific features or interact with hardware components. To interact with privileged components, Services interact with the `system_server`, which runs as the privileged system user. A vulnerability in some Service APIs can lead to the disclosure of sensitive information opening the door for state inference techniques. Many works [14], [38], [47] investigated these aspects showing many vulnerabilities of the Android components. For instance, Possemato et al. [38] detected several vulnerabilities, among which one that allows an app to collect statistics of other apps through the `getProcessMemoryInfo` API of the `ActivityManager` without any special permission.

To mitigate these attacks, Google has deployed several patches and countermeasures. First, Android strengthened the data returned from the APIs removing all the sensitive information of other apps. For instance, the `getRunningTasks` and `getRunningServices` APIs of the `ActivityManager` class returned the list of the tasks and services currently running in a device. These methods have been deprecated from respectively Android versions 5.0 and 8.0; nowadays, they return only the information of the caller app (and well-known non-sensitive data). In Android 7.0, access to `procfs` is limited to very few harmless files. As a result, to the best of our knowledge, all documented file system-based state inference attacks no longer work on modern versions of Android. However, despite these mitigations, some recent works have shown that phishing attacks are still feasible [6], [10].

On the defensive side, some works have been proposed over the years. [33], [39], [55] tried eradicating the phishing problem on Android, preventing a malicious user from spoofing the UI. The authors do not avoid a malicious app to perform the state inference attack on which the phishing is based but block or react to malicious UI that seems legitimate. Only "LeaveMeAlone" [58] tries to detect and block a malicious app that performs inference attacks (or tries to steal sensitive data) by analyzing the usage of shared resources. This tool leverages static information (e.g., permissions) and the `procfs` files to monitor the device apps, but it will not work on Android 7.0 and newer due to the enforcement of the `procfs`.

Finally, in 2016, Ahn et al. proposed *inishing*, an inotify-based user interface (UI) phishing attack, which at the time of writing is the work most closely resembling ours. They show how malware can monitor *access patterns* (i.e., a sequence of file system events), enabling it to detect the transition from one UI screen to another. Once the access pattern events are triggered during a screen transition, the malicious app needs to check whether the target app generated it. The authors retrieved the PID of the target app through the `getRunningAppProcesses` API (assuming it is already running). They verified whether it is in the foreground by inspecting the files in the `procfs`. However, since Android 7, Google has significantly restricted access to the `procfs` folder, and only system apps can obtain the list of the running processes (and their pid). These restrictions make this attack infeasible, but as we will show, there is still room to abuse inotify.

4. The use of inotify among malware

To verify the novelty of our attack, we investigated if and how current malware exploits inotify. To this aim, we collected and analyzed 10,000 Android malware of 2021 from AndroZoo [4]. We considered malicious the samples with at least five anti-malware detections on VirusTotal.

For this type of study, it is crucial to distribute the malware families as uniformly as possible; therefore, we considered a maximum of 5% for each family. Given that Androzoo does not indicate the family of its malicious samples, we downloaded each sample's VirusTotal report and leveraged AVClass2 [44] to determine the corresponding family. Finally, AVClass2 classified a total of 140 families and 8400 samples as *Singleton* – the term used to define the samples for which it was unable to determine the family. We report the percentages of the five most frequent families and the Singletons in the second row of Table 2 in Appendix A.

As explained in Section 2.2, an app can install an inotify watch from both the "Java" and the native layers. To account for either case, we developed two software components to analyze Android apps statically. The goal of these components is trying to resolve the arguments – performing a backward iter- and intra-procedural taint data analysis – of the API functions that interact with inotify to determine the path of the file that will be monitored.

Java layer. The first component is an extended version of the Soot [52] framework to analyze the DEX files and resolve the constructor's argument of the `FileObserver` objects that takes in input the path of the file to monitor. When the tool resolves a `java.io.File` object, it performs a backward taint analysis until it reaches an Android API method invocation (e.g., `getExternalFilesDir`) or a string. This procedure is repeated recursively if it finds another `File` constructor. The output of this component may be a string with the full file path or a combination of Android-specific APIs. For instance, if an app installs an inotify watch on the file named `example` in its folder on the external memory, and it uses the `getExternalFilesDir` method (of the `android.content.Context` class) to retrieve the path of the external memory, the output is the concatenation of the Android API and the file name: `android.content.Context.getExternalFilesDir()/example`.

The analysis reveals that 7.4% (744/10,000) of our malicious apps, distributed over 65 different families, instantiates `FileObserver` objects. Among the 744 samples, we measured 57.8% (430/744) Singleton, and we did not observe a predominant family.

Most resolved paths depend on Android APIs to return the path of "private" folders, where the app can place persistent files it owns. We listed them and their prevalence

in Appendix D. Given that in almost all these cases, the malware installs a watch on a file under these folders, there is no particular interest in monitoring files of other apps. Manually looking at folder and file names, we found that these cases are mainly analytics libraries. For instance, the Yandex [8] analytics library monitors the crashes of an app watching the file `appmetrica_crashes` in the app's file directory (i.e., `Context.getFilesDir()/-appmetrica_crashes`). The most common target is the folder `/data/anr`, which was detected in 22.2% (165/744). When the UI thread of an Android app is blocked for too long, Android stores logs of the "App Not Responding" (ANR) errors in that folder.

The only pattern we found where the samples seem to monitor file system events generated by folders shared with other apps was present in the 4.2% (31/744) of the samples across eight families – disregarding the Singletons, the most prevalent (5/31) is the trojan `triada`. Such samples retrieve the path of its folder in the external memory and install an inotify watch for all the parent folders. However, this code belongs to the Vungle [53] library for advertising and monetization.

Finally, in some cases, the tool could not recover the path because I) the sample concatenates strings at runtime (26.7%), II) it is passed through an (external) Intent (0.3%), or III) it is read from the shared preferences (0.1%). Therefore, we manually reverse-engineered some samples, one for each family, and then moved to the Singletons. During this manual investigation, we did not find any new malicious pattern.

Native layer. The second component is a custom Ghidra [1] plugin, and it analyzes the native libraries to identify the calls to the `inotify_add_watch` API. In the malware dataset, 87.4% (8,742/10,000) of the samples contain native code. The analysis revealed that 77.0% (6,731/8,742) of the malware samples use the inotify API in just 339 native libraries. Given that we did not find a predominant family (third row of Table 2), this low number of libraries w.r.t. a high number of samples is because these samples share such libraries. Most are analytics libraries, such as Umeng [57], that monitor files in their private directory to identify app failures.

The analysis of native libraries proved to be more complicated than the Java one; our tool failed to resolve the argument to the `inotify_add_watch` function in more than 40% of cases because the string was not statically available. Therefore, we resorted again to manual reverse engineering, using the same strategy discussed above, stopping when we did not notice new patterns emerge. With this mixed manual and automatic approach, we uncovered two "malicious" cases. First, an anti-debug strategy that we often, but not only, found implemented in the Jiagu packer. To give a practical example, we analyzed one sample[1] that creates a new thread, and from this thread, it uses inotify to monitor the `mem` and `pagemap` files in the `/proc/<pid>` folder. In this way, it checks in real-time a change in the memory mapping of the app process to detect the presence of a debugger.

Second, we observed some apps that monitor if specific files inside their installation folder are deleted (e.g., the `base.apk`). In this way, they can identify if the app is about to be uninstalled and run extra code; for instance, some samples[2] open a web page. Interestingly, this technique is the one that comes closest conceptually to what we found in our results, and this shows again that allowing to monitor files within the installation path opens the door to state inference.

RQ1. Malware uses inotify in both Java and native code, but in most cases, we observed non-harmful operations, such as reacting to an app crash. We have encountered this phenomenon because malware is often produced by repackaging popular apps with malicious components. Therefore, we mainly detected the use of inotify by analytics libraries that can also be found in goodware. Only 4.2% of the samples try to monitor file system events generated by folders shared with other apps, but this behavior is again related to a famous advertising library. We found no samples installing an inotify watch on a file inside the installation folder of another app; in general, we found no cases suggesting the use of inotify for a state inference attack. However, we found two interesting cases in native code where malware abuses inotify: an anti-debug technique and a "last resort" to execute code when they are uninstalled.

5. Analysis Tool

While comforting from an end user's perspective, the fact that malware does not seem to leverage inotify for state inference attacks poses interesting questions. For example, have malware authors never come up with this attack idea, or do they instead not implement it because it does not work in practice?

While answering such questions on malware authors' behalf is rather tricky and far beyond the scope of this paper, we can evaluate the feasibility of employing inotify for state inference at scale.

For this purpose, we designed *inoTool*: a tool, written in about 2k lines of Python, that can prove whether an Android app is vulnerable to inotify-based state inference. *inoTool* exposes three functionalities that capture and analyze an app's file system behavior as discussed in the remainder of the section.

5.1. Function #1 – *logEvents*

This function takes as input an APK file and computes the list of the file system events (we will refer to this list as *FSFootprint*) generated by the APK during its execution. We represent a file system event as the pair ($event\_type$, $path$), where $event\_type$ is the inotify event corresponding to the file system action performed on the file at $path$ (e.g., the event `IN_MODIFY` encodes a file content modification). Furthermore, *inoTool* stores the mappings between the app's package name and its *FSFootprint*.

At first, *inoTool* installs the app to analyze and runs it for the first time without any monitoring. The idea behind the first "dry-run" of the app is to avoid capturing file system activities that are just the result of some post-installation setup operations. For example, an app that uses an SQLite database stored on the device's file system. The app likely creates this database only once,

1. SHA-1: `9f02e7edbbf9aa8874fc4e93dbb50955d52793ee`

2. SHA-1: `a1604d3705511099a75bc522421c68bb450140f6`

when it starts for the first time. If we considered the first run representative of the subsequent execution, we would wrongly expect it to create the SQLite file each time.

It is only while running the app a second time that *inoTool* collects the file system behavior of the app over a time span of 5 minutes. During this second run, *inoTool* also stimulates the app user interface, thus increasing its code coverage. For this purpose, we used ARES [41], a black-box tool that uses Deep Reinforcement Learning to test and explore Android apps.

We developed our file system events monitor as an eBPF program that hooks all the system calls (*syscalls* from now on) related to file system events that can generate an event. We reported the complete list of monitored syscalls in Table 3 in Appendix. eBPF (extended Berkeley Packet Filter) [29] is an in-kernel virtual machine that provides an interface to extend the capabilities of the Linux kernel with user-provided code. It has been available in Linux since version 4.4 and provides two advantages compared to adding new code directly in the kernel. First, the eBPF code can be updated without recompiling the entire kernel. Second, the eBPF VM code does not support potentially dangerous constructs (e.g., loops with non-fixed iteration counts) and undergoes a strict verification phase before execution, making it less prone to bugs. eBPF programs are event-driven: the eBPF code is executed only in response to specific events that include syscalls, making it a powerful tool for the syscall-based monitoring of Android apps.

The eBPF program is written in about 1,100 lines of C and leverages kernel `tracepoint` and `Kretprobe` subsystem for monitoring syscalls. Tracepoints allow the eBPF program to intercept the beginning of each syscall and parse its input parameters. Kretprobes, instead, are used to monitor the return values and track the opened/closed files for each process, which is crucial because it allows *inoTool* to interpret syscalls that accept file descriptors as input parameters. For instance, the first parameter of the `read` syscall is the file descriptor of the file from which to read. Our eBPF program keeps track of the mapping between file descriptors and file paths for each process in the system during the app's runtime under analysis.

To start and stop the eBPF program each time a new app runs, we created a second program that registers/unregisters the kernel hooks and queries the eBPF program to retrieve the recorded events and save them into a JSON file. The second program is written in Golang (about 700 lines) and leverages the `libbpfgo` library [45] for interacting with the eBPF programs from userspace.

Finally, the function analyzes the syscalls recorded so far and converts them to a list of file system events (i.e., *FSFootprint*). While, in most cases, mapping a syscall to an inotify event is straightforward, we had to cope with some special cases. The interested reader will find all the details in Appendix B.

5.2. Function #2 – *generateSignatures*

This function takes as input a set of apps K (cardinality $|K|$), and their respective *FSFootprint* generated by *logEvents*. It returns the *FSSignature*$_K$ for each app, that is the set of file system events that our tool recorded **uniquely** for the startup of that APK; namely, it considers all the unique startup events of that APK's *FSFootprint* w.r.t. the other APKs in the set K. Notice that the choice of the apps in K determines how reliably an attacker can use the *FSSignature*$_K$. By tailoring K to the apps installed on the device, she is targeting the attacker can ensure that a file system event uniquely identifies an app's startup on that device.

Determining an app startup. Since our attack aims at identifying an app startup, while constructing the *FSSignature*$_K$, we want to consider only those events generated during the startup phase. However, determining when an app startup ends presents several challenges. Naive approaches like monitoring the screen until the user interface stops changing are not an option because apps showcase very different UI when they start (e.g., some have long-loading screens with animations, while others show advertisements). Relying on triggers in the app's code is also not feasible due to the heterogeneity in the UI frameworks Android apps use (e.g., Cordova, Xamarin, Flutter, WebView, etc.). For the purposes of this work, we resorted to an experimental human-in-the-loop approach to determining when an app finishes loading. In particular, we randomly selected and ran 100 apps in our emulated environment while recording the emulator's screen. We then manually inspected the recordings and marked in each of them the time at which, according to our human understanding of the app's context, the startup process ended. The minimum startup time (rounded down) we measured was four seconds, which we then used to mark the end of the startup process for all the apps. We chose the shorter time conservatively to ensure we do not consider already started apps.

More precisely, the *FSSignature*$_K$ (S) of the n-th app is computed as:

$$S_n = F_n^+ \setminus \bigcup_{i \neq n} F_i \qquad \forall e \in F_n^+,\ t_e \leq 4\ seconds$$

where F_k represents the *FSFootprint* of the k-th app, and F_k^+ its subset that contains only the startup events (i.e., events that occurred in the first four seconds). It is worth noticing that to compute the *FSSignature*$_K$, from the F_k^+, we filter out all the events in the *FSFootprint* of the other apps in K, regardless they occurred at the startup or during the apps' runtime. This procedure allowed us to avoid the case when a file operation performed by app A at its startup is also performed by app B at its runtime. Therefore, if an app has a non-empty *FSSignature*$_K$, monitoring one event of its signature is a good indicator to infer when it is starting up.

5.3. Function #3 – *signatureVerifier*

By construction, the *FSSignature*$_K$ of an app contains those unique events recorded during its startup only; i.e., the same events did not occur during other apps' execution. This means that an attacker capable of intercepting any event in an app's *FSSignature*$_K$ can mount an unambiguous state inference attack, using such events as an oracle.

This function, which takes as input an APK and its *FSSignature*$_K$, aims to assess whether it is indeed vulnerable to inotify-based state inference attacks or, in other words, whether an attacker can leverage an app's *FSSignature*$_K$ to mount a state inference attack. We accomplish this by mimicking an attacker's behavior using

an *a posteriori* approach: we refine the $FSSignature_K$ using the events we can confirm that can be used in the attack. Thus, the result of this function is $FSSignature^*_K$, a subset of the $FSSignature_K$. In the end, if the $FSSignature^*_K$ is not empty, we declare such an app vulnerable.

We implemented our attack in an Android app we named *appSignVerifier* that attempts to intercept any event in an app's $FSSignature_K$. Under the hood, *appSignVerifier* uses inotify through the `FileObserver` class. Before registering an inotify watch, *appSignVerifier* checks whether the target file exists. If that is the case, the watch is registered directly on the file. In contrast, if the file does not exist, *appSignVerifier* registers the inotify watch on its parent directory to intercept the file creation.

The entire verification stage adopts the following workflow. Similarly to *logEvents*, the *signatureVerifier* installs the app under test and starts it for the first time. It then provides the app's $FSSignature_K$ and its installation path to the *appSignVerifier*, which sets an inotify watch on the events in the app's signature. *appSignVerifier* needs the installation path because events in the $FSSignature_K$ may be related to files in this directory; however, this path may change across two installations. The function starts the target app for a second time, allowing the *appSignVerifier* to collect the inotify events. After the app startup, it stops the *appSignVerifier*, which dumps the collected file system events on mass storage. Then, it prunes the list of recorded file system events of any entry not in the $FSSignature_K$. This allows to filter out those events that the *appSignVerifier* accidentally recorded while monitoring *any* file creation event in a directory[3].

At the end of the analysis, *inoTool* produces three sets of file system events for each app, for which the following relationship holds: $FSFootprint \subseteq FSSignature_K \subseteq FSSignature^*_K$. The cardinality of an app's $FSSignature^*_K$ equals the number of file system events in its $FSSignature_K$ that our *appSignVerifier* was capable of intercepting. Therefore, an app's $FSSignature^*_K$ is not the empty set if and only if at least one of its file system events arises during its startup only (that is, the same event does not take place during any other app's execution in a set K) *and* an untrusted app can effectively intercept such event.

6. Experimental Evaluation

This section presents the setup under which we tested a dataset of 4,863 Android apps using *inoTool*, as well as a preliminary and coarse-grained analysis of our findings.

6.1. Experimental Setup and Dataset

Our analysis system is based on the Android emulator and runs Android version 12.0 (API level 31), equipped with the latest Google Play APIs, thus implementing all the latest security features. We equipped the emulator with a dual-core 2.10 GHz x86-64 processor, 2 GB of RAM, 32 GB of internal storage, and a 16 GB emulated SD Card.

We built the dataset of apps to test by collecting Android applications that are valuable and widespread among Android users and that implement a relatively heterogeneous set of functionalities. The first two characteristics make these apps appealing targets for attackers, which may profit from a vast user base. On the other hand, the variety of functionalities ensures that the app we analyze shows different behaviors. This prevents our results from being skewed toward one particular type of app and more representative of the average user's device. As reported by Google [24], Android users install, on average, at most 35 apps spread over different categories.

We collected the 100 most downloaded free apps for the 50 categories in the Google Play Store [19]. Among these categories, some are particularly valuable, especially for phishing attacks, such as finance and social media apps. In total, we could install 4,863 apps because some of them were incompatible with our emulator.

3. For example, if the $FSSignature_K$ of an app contains the path `/example/uniq.txt`, but this file does not exist before the app starts, *appSignVerifier* monitors the folder `/example/`, in which, however, other uninteresting file system events may occur. These events should, thus, be discarded.

Table 1: Cardinality statistics with $|K| = 4{,}863$

Set	Min	Max	Avg	Stdev	Median
FSFootprint	17	3087	463.2	192.1	457
$FSSignature_K$	3	784	65.1	74.2	28
$FSSignature^*_K$	2	61	6.8	2.6	7

6.2. Preliminary Analysis

Table 1 shows aggregate data about the *FSFootprint*, $FSSignature_K$, and $FSSignature^*_K$ that *inoTool* generated for each app.

On average, the cardinality of an app's *FSFootprint* is seven times that of its $FSSignature_K$. This means that, as one may expect, most of the file system activity during an app's startup does not depend on the app itself but rather on the implementation of the app startup process. For example, to bootstrap any apps, the Android system must load and read the `/system/framework/x86_64/boot-framework.art` and `/apex/com.android.art/-javalib/x86_64/boot.art` files, which contain the ahead-of-time compilation result of some framework classes. Nonetheless, roughly one in four file system events are unique to the app, leaving room for inferring which app is starting based solely on the file system activity.

While significantly (roughly ten times) smaller than the $FSSignature_K$, the $FSSignature^*_K$ of an app contains, on average, 6.8 file system events. As a reminder, these are file system events that are not only specific to the app but can also be monitored by a third-party application using the `inotify` system. Much to our surprise, the lowest cardinality among the computed $FSSignature^*_K$ is two, meaning that *all* the apps in our dataset have a non-empty signature and are susceptible to inotify-based state inference attacks.

Upon further analysis, we found some file system events common to the startup process of all the apps we tested. Since they stem from the app startup process implemented by the Android system, we define these events as *system-dependent*.

An example of a system-dependent file system event is the opening of the `base.apk`. This file is located in the app's installation directory. It is accessed by the Android OS to retrieve its resources (e.g., the Manifest), resulting in an `IN_OPEN` and an `IN_ACCESS` file system events. Our analysis demonstrated that *any* program in the system

that knows an app's installation path could intercept these two events on such a file. This is possible because of the permissive access control rules applied to this file. From a DAC point of view, any app's `base.apk` is world-readable, while SELinux MAC rules explicitly allow it (discussed in Section 8).

Other system-dependent events concern the `base.odex` and `base.vdex` files, which are also located in the app's installation path. These files are the byproduct of the ahead-of-time compilation process implemented in the Android RunTime that translates the app bytecode into machine code at installation time. As such, the Android framework has to access and load these files in the address space of the newly spawned app; in fact, the former contains the machine-compiled code, while the latter stores data designed to speed up execution.

App-dependent events represent the other side of the coin compared to system-dependent ones. These file system events are tightly related to an app's behavior rather than the implementation of the bootstrap process in the system framework. App-dependent events may concern files located anywhere on the file system. In Appendix C, we reported the complete distribution of the app-dependent events throughout an Android file system.

Although not as common as their system-specific counterparts, the most common location of app-dependent events is still the app's installation directory (`/data/app` row in Table 4), in which 17.2% of apps generate at least one event. Examples of such events are those related to native libraries shipped with the app and placed in the `lib` directory in the installation path.

1.5% of apps generate at least one event in the `/storage/` top-level directory – also known as *external storage* – which contains app-specific sub-directories where apps can place their persistent files. Notably, external storage comprises two parts. The first is a mount point at which the Android system mounts external mass memory devices (e.g., SD cards), which account for roughly 45% of the events *inoTool* detected in the external storage. On the other hand, the second is not backed by any external memory device and is created by the system to provide external storage functionalities without external memory support. This second part accounts for most of the events *inoTool* reported in external memory.

File system events in other subdirectories are generally less common and affect less than 1% of the total apps in our dataset. Nonetheless, an attacker can leverage them to mount for state inferencing purposes. Interesting examples of events in these paths concern the `/system` top-level directory. A peculiar case is that of an app that interacts with the `/system/priv-app/InputDevices/InputDevices.apk` file to check if a particular input device is available.

7. Attacker Models & Attack Scenarios

Analyzing the results obtained by testing the 4,863 apps through *inoTool*, we identified three distinct sets of capabilities that an attacker needs to perform an inotify-based state inference attack. In this context, a *capability* may be an Android permission or any information about the installed apps. In general, obtaining a specific capability may depend on I) the version of the Android system and II) the API level that the malicious app targets. The impact of the former on the attacker's capability is relatively intuitive: different versions of Android implement slightly different functionalities and security models. To fully appreciate the latter's role, one needs to consider how Android deals with the app's backward compatibility. At compilation time, an app developer can choose which version of the Android framework the app is intended to run. If the targeted version is "too old," the OS will refuse to install the app. On the other hand, if the targeted version is still supported, the system emulates the targeted version's behavior and security model. For instance, more than 68% of the apps in the dataset target API 30, even if, at the time of writing, the latest Android version is 13 (API 33). As such, most of the apps we analyzed are not subject to the latest and more restrictive security model.

In the remainder of the section, we will assume that the attacker has paved her way to install her malicious app on the victim's device – a basic premise for any state inference-based phishing. While this may happen in various ways and represent an interesting research field in its own right, we will not develop this topic further as we consider it orthogonal to the research questions this work addresses. Notice, however, that we assume the malicious app to be a regular Android app, thus subject to the Android security model for *untrusted apps*, with restrained access to the device's resources.

7.1. Attack #1

As we saw in Section 6, while starting any app, the Android system accesses the app's specific files (e.g., the `base.apk`) in its installation directory. In other words, an attacker able to monitor those files can target *any* app on the system. Luckily, an app's installation path is not known a priori (part of its name is randomly generated at installation time), and, at least in theory, it can only be retrieved through the *PackageManager* service. Before version 11, interacting with this service did not require any permission, allowing all apps on the device to access each other's installation path. To limit Software Discovery [12] (i.e., listing installed apps and their metadata), Android 11 introduced the concept of *package visibility* [21]. In practical terms, under the new security model, an app must either list the metadata of all third-party apps it needs to access (through the `<queries>` tag in its Manifest) or request the `QUERY_ALL_PACKAGES` permission. From a malicious actor's perspective, neither of these mechanisms is ideal. For what concerns the first option, an app whose Manifest targets many apps is rather suspicious and incurs higher risks of being banned by the marketplace. Similarly, requesting the `QUERY_ALL_PACKAGES` permission is not a sustainable plan for an attacker. In fact, Google states that any app requesting this permission must undergo manual scrutiny before being accepted on the Play Store [21].

However, there are two loopholes that a malicious app can leverage. The first takes advantage of Android's backward compatibility system. The *package visibility* mechanism does not apply to apps targeting API level 30 or lower, which can retrieve the installation path of *any* app in the system without any permission. The second technique consists in (ab)using one feature of the `<queries>` tag that gives access to any app that

implements a particular *intent filter* [26]. The ruse consists in specifying an intent filter, as shown in Listing 1.

Listing 1: `queries` tag to interact with all applications

```
<queries><intent>
 <action android:name="android.intent.action.MAIN"/>
 <category
      android:name="android.intent.category.LAUNCHER"/>
</intent></queries>
```

This allows querying *all* the apps with a "launchable" activity – i.e., those presenting one entry in the Android launcher UI – which is the case for practically every commonly used app. While conforming to the documentation, this stratagem contradicts the app separation principle at the core of the Android security model. We also reported this bug to Google, which acknowledged its security implications and said that the issue had already been reported but not yet made public.

7.2. Attack #2

The second attacker model we envisage assumes that the loopholes we introduced in Attack #1 are no longer available, making monitoring the apps' installation path impossible for an attacker.

In this new scenario, our attacker chooses a dataset of apps as large and heterogeneous as possible and similarly analyzes them to that presented in Section 6, embedding each app's $FSSignature^*_K$ in her malicious app. In analyzing what an attacker can achieve by implementing this strategy, we will refer to the data we collected using *inoTool*. The files in the $FSSignature^*_K$ generated in our experiments can be divided into two categories: those that can be accessed via the `READ_EXTERNAL_STORAGE` permission, and those that are world-readable, i.e., they do not require any permission at all to be monitored.

`READ_EXTERNAL_STORAGE` permission. Earlier Android versions used this permission to restrict access to the *external storage*. Android 10 (API 29) introduced a new file system paradigm called *scoped storage* [23] that divides the external storage into private and shared portions. As the names suggest, while the former is app-specific, the latter contains data shared across all the apps in the system and is further divided into media and non-media content. The new paradigm redesigned the `READ_EXTERNAL_STORAGE`, allowing an app to read only other apps' shared media content. Consequently, it drastically changes the benefit an attacker gains by obtaining the permission in question. While on older versions of Android (API < 29), this permission allows identifying correctly apps that access any file in the external storage, on newer versions, it only allows detecting those that access shared media files.

Based on our dataset, this discrepancy translates into roughly ten times fewer apps being vulnerable while running in Android 10 or newer (0.7% of the dataset) w.r.t. older versions (18.1%). These figures do not include world-readable files (discussed below); as such, they can be considered the permission's actual contribution.

World-readable files. We discovered something peculiar by analyzing the world-readable files *appSignVerifier* managed to exploit. While on paper, files in the "private" storage of an app are accessible exclusively by the app itself, in practice, it is still possible to monitor some of them through *inotify* (Appendix E provides insights on the circumstances under which this is possible).

To make things worse, unlike the installation directory, an app's private storage path does not contain any randomly-generated part, making it trivial to infer. In other words, in a bizarre turn of events, the scoped storage's implementation breached the tight access control that regulated the external storage in previous versions of Android. In our dataset, 0.9% of apps are vulnerable to state inference attacks that do not require any particular capability from the attacker. Among such apps, we find two famous instant messaging (WhatsApp and JioChat) and a money transfer (Venmo) app. They have large user bases, having been downloaded at least 10M times each; therefore, unauthorized access to user accounts on these platforms may have severe privacy and financial repercussions.

7.3. Attack #3

The main advantage of the attack strategy we just described is that the pre-computed $FSSignature^*_K$ can be embedded in a single malicious app and shipped *as is* to the victim's device. This strength, however, comes at the expense of overapproximating what the vulnerable applications are on an actual device. When computing the file system signatures on a dataset K as large as the one we used in our experiment, the attacker implicitly assumes that *all* the apps in the dataset could be installed at once on the targeted device. This is obviously not the case for the average Android device that, according to Google [24], tends to have around 35 apps installed at a time.

Intuitively, narrowing down the cardinality of K, increases the number of apps with a non-empty $FSSignature^*_K$. The sweet spot for an attacker is to choose K as the set of apps installed on the target device. Doing this, in fact, maximizes the number of apps that are vulnerable to state inference attacks *on that specific device*.

Thus, the third attack strategy we introduce consists in scouting the apps installed on the infected device, computing the apps' $FSSignature^*_K$ accordingly with the aid of a remote endpoint, and employing them to perform state inference attacks.

The sole capability the attackers must acquire to implement this strategy successfully is obtaining the list of installed apps on the device. Similarly to what we described in Section 7.1, nowadays, the most trivial way to do that would be through either the `QUERY_ALL_PACKAGES` permission – which is too suspicious to be considered a viable option – by defining the `queries` tag or targeting API level 29 or below. One last option at the attacker's disposal is leveraging an information disclosure vulnerability, like the one we discovered in our research.

Information Disclosure Vulnerability. The introduction of scoped storage loosened the access control on external storage, allowing otherwise unprivileged apps to mount state inference attacks. In particular, we noticed that the paths where each app saves its private files are predictable, following the pattern `<externalstorage path>/Android/data/<packagename>`. For example, on our test device, the private files of

the famous Discord app (whose package name is `com.discord`) were located in the `/storage/emulated/0/Android/data/com.discord/` directory. Unlike the app's installation path, the private storage path does not contain any aleatory component, making it trivial for an attacker to infer. Simply invoking the `getExternalStorageDirectory` API (which does not require any permission), an app can retrieve the `<externalstoragepath>`, while the target app's package name can be determined by statically analyzing its APKs available on the marketplaces.

What makes private storage vulnerable, however, is the fact that the app's files are at predictable locations *and* the permissive access control rules that the system applies to them are not enough. The `data` directory in the external storage is marked as not readable, not writable, but executable for any untrusted app. Under the POSIX convention, this combination of flags denies anybody but the directory's owner and group to list its content and create files in it. However, the executable attribute allows anybody to *traverse* the directory. Therefore, an untrusted app – even *without* the `READ_EXTERNAL_STORAGE` permission – can check if a specific file exists in the `data` directory using of the `newfsstat` syscall or, more conveniently, through the `java.nio.file.Files.exists` Java wrapper. More specifically, when querying for an existing file, the syscall (and the wrapper) returns successfully. To an extent, this primitive can circumvent the lack of listing privilege on the directory. Thus, an attacker can maintain a list of package names of apps they want to target and systematically probe each of them.

If such a folder is in the `data` directory, the attacker is sure that the corresponding app is installed on the device because all its private folders are removed whenever an app is uninstalled. In addition, the system creates the app's private folder on the external storage whenever it invokes one of the several methods (complete list in Appendix D) to retrieve its private location for the first time.

By statically analyzing the apps in our dataset, we discovered that almost the entire dataset ($> 99\%$) imports at least one of these methods. In particular, the `android.content.Context.getFilesDir` method is the most prevalent and can be found in 98.7% (4,632/4,863) of the apps in our dataset (see Appendix D for the full list). However, the static analysis does not guarantee that every one of these apps invokes any of these methods. This result should be considered an upper bound of the share of apps that use external storage and are thus vulnerable. To estimate a lower bound, we measured the number of apps for which *inoTool* registered a file system event in the corresponding private folder, and we obtained 49% (2,383/4,863).

In conclusion, we can estimate that 49-99% of the apps in our dataset can be detected on a device without permission. This information disclosure vulnerability partially thwarts Google's efforts to restrict third-party apps from Software Discovery.

Performance of the refined attack To estimate the likelihood of success of an attack tailored to the apps installed on a device, we relied on random sampling apps from our dataset and computing the number of vulnerable apps in each case. For each simulation, each app's chance to be selected depends on the downloads from the Play Store, so the ratio between the two apps' download counts equals the ratio of their chances to be selected. For example, the eToro and FlyerMaker apps have been downloaded by 10M and 1M users, respectively; thus, the former is ten times more likely to be selected than the latter.

We varied the cardinality of the sampled apps from 15 to 100, increasing it by five each time. We created $5,000$ unique app sets for each cardinality, i.e., $5,000$ sets of cardinality i. For each set, we executed the *generateSignatures* and *signatureVerifier* functions of *inoTool* to measure the number of vulnerable apps to our attack with this particular device configuration.

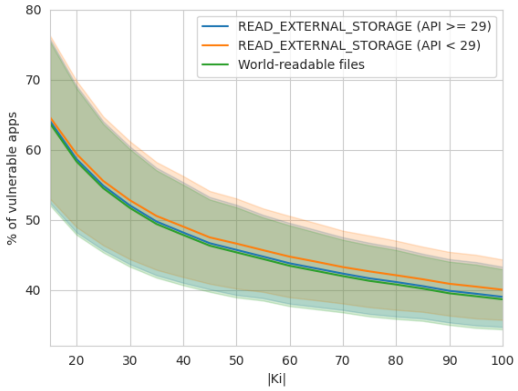

Figure 1: Capability for different K_i values

Figure 1 shows the results of this experiment, divided according to the permissions and Android version required to monitor files with inotify. The lines are averages of the percentages of vulnerable apps, while the faded areas are the standard deviation.

As expected, the number of vulnerable apps decreases with the number of apps installed. Alarmingly, on devices with an average number of installed apps (35, according to Google [24]) running modern Android versions, an attacker can successfully target half of the apps, even without declaring any permission. While performing slightly better when targeting API level < 29, under modern security models, the `READ_EXTERNAL_STORAGE` does not significantly affect the likelihood of carrying out the attack w.r.t. to targeting world-readable files only.

7.4. Final Considerations

The following is a summary answering **RQ2** and **RQ3**. We identified three types of attacks, which differ according to the attacker's capabilities. In case an attacker can monitor a system-dependent file system event (Attack #1), the opening of the `base.apk` is the best candidate. To this aim, she needs to know the installation path, and the only ways to get it are: by requiring the `QUERY_ALL_PACKAGES` permission, abusing the `queries` tag, or targeting SDK ≤ 29. This attacker is the most powerful because it works with 100% of apps regardless of the Android version and the device configuration.

In case an attacker cannot retrieve the installation path, she can pre-compute a $FSSignature^*_K$ with a large K analyzing as many apps as possible to find peculiar

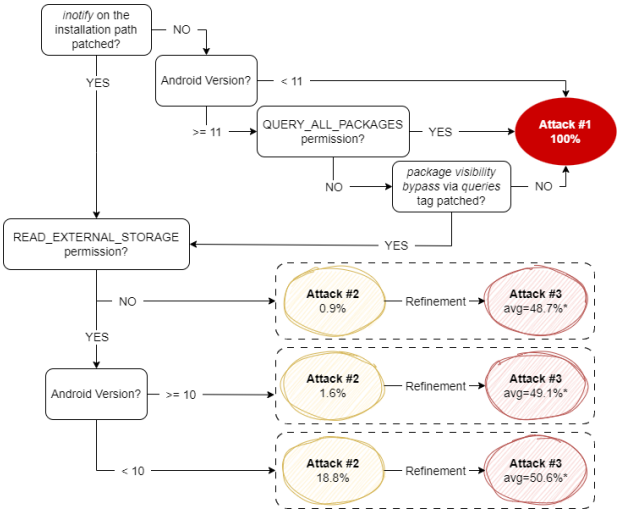

Figure 2: Flowchart summarizing the attack models

behaviors that enable an attacker to perform an inotify-base state inference attack (Attack #2). Our evaluation with $|K| = 4{,}863$ found that an attacker can target 0.9% of the app by monitoring only world-readable files. Obtaining the `READ_EXTERNAL_STORAGE` permission, the number of vulnerable apps increases to 1.6% and 18.8% for Android 10 or newer and older versions, respectively. However, these values also consider the contribution of the world-readable file since an attacker can always monitor them. Thus, the actual contribution of this permission is 0.7% on Android 10 or newer and 18.1% on older versions. Despite these low values, this is the most general scenario: an attacker can always attempt to use this approach to search for peculiar events in the *FSFootprint* of the target app.

Attack #3 is a refinement of the previous attack strategy that an attacker can adopt if she can get the list of installed apps, e.g., by using the information disclosure vulnerability we reported. Our experiments suggest that in a typical device with 35 apps installed on average [24], the attacker can target almost half of the apps without permissions.

Finally, the Android version largely influences the number of vulnerable apps and the attacker's capabilities. First, the metadata of the installed apps (e.g., the installation paths) can be obtained without any permission on a device running Android < 11; on Android ≥ 11, an attacker has to update the Manifest file appropriately. Second, the number of apps vulnerable requiring the `READ_EXTERNAL_STORAGE` permission strongly depends on the version of the OS.

We summarized the workflow of the different attacks in the flowchart in Figure 2, and for each case, we report the percentages of the vulnerable apps in our dataset. The percentages reported for Attacker #3 consider the average case of the typical device with 35 apps installed, as described above.

7.5. Use Case with Contextual Notification

To demonstrate the impact of an inotify-based state inference attack, we developed a toy app implementing a full zero-permission phishing-like attack targeting the eToro (`com.etoro.openbook`) app. According to the Play Store, this app counts tens of millions of installations and provides access to a trading platform, making it the perfect target for a malicious actor.

The $FSSignature^*_K$ we computed for this app contains the file system events for opening (`IN_OPEN`) and reading (`IN_ACCESS`) the `/system/framework/-android.test.runner.jart` file. The corresponding file is accessible by any app on the system, including unprivileged apps, without requiring any permission.

Figure 3 shows the attack flow in three steps. Our malicious app registers an inotify event at the beginning of its execution, using the `FileObserver` API When the user launches the eToro app (Step 1 in Figure 3), the inotify callback is triggered. At this point, our malicious app does not immediately create an activity that mimics the eToro's one. In the modern Android system, in fact, starting an activity without the user's interaction is strictly regulated since this type of action has long been used to implement UI hijacking. The only viable way to implement such behavior is by obtaining the `SYSTEM_ALERT_WINDOW` permission, which has been ranked as *privileged* after previous works demonstrated its malicious potential [16]. This fact did not stop malware authors from using it; e.g., 56.9% of the samples (5,692/10,000) among the malware we used in this study (Section 4) require it, while in the case of benign apps (Section 6) 8.6% (431/4,863). However, implementing our phishing attack using this permission defeats our purpose of showcasing how a complete permission-less phishing attack.

Xu et al. already proposed using the notification service on a mobile phone to launch phishing attacks [56], and we decided to refine this idea. Instead of requiring the abovementioned permission, our malicious app opts for displaying a *contextual notification* (Step 2 in Figure 3), trying to lure the user into initiating a fake update process for their eToro app. A contextual notification, while unexpected, is not necessarily suspicious from the user's perspective. In this case, the app may have found an available update while loading and asked permission to install it.

Since it mimics the target's look&feel, for the user is very difficult to distinguish our malicious update prompt from a regular eToro notification. Step 2 also shows how legitimate and phony notifications render when the malicious app targets different Android operating system versions. In the first case targeting an older version of Android, the forged notification is practically impossible to distinguish from a legitimate one because the attacker has complete control of the UI. While targeting Android 12, instead, the user's only chance to understand that the notification is not genuine involves expanding the notification[4]. As the Figure shows, under these circumstances, the user can read the name of the apps that generated the notification – which is, nevertheless, attacker-controlled, as we emphasize in our example.

Once the user clicks on the contextual notification (Step 3 in Figure 3), our malicious app can effectively take control of the UI system and finally creates the activity that asks the user for their credentials.

RQ4. During our analysis, we discovered several loopholes that grant an attacker different capabilities to perform

4. Starting with Android 12 (API level 31), apps are no longer able to create fully custom notifications [25]. The system applies a standard template instead.

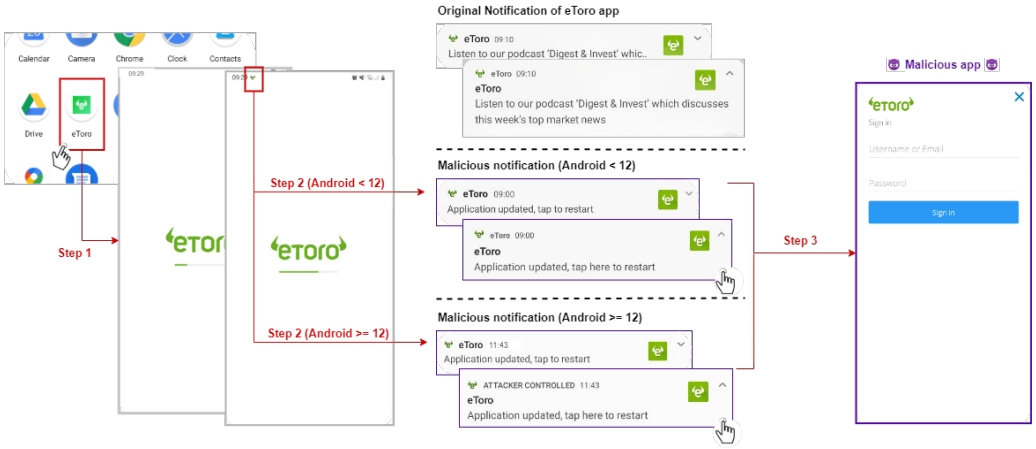

Figure 3: Flow of the attack with a contextual notification

a state inference attack and mount a practical phishing attack. Moreover, we highlighted that a malicious actor can still perform this attack without particular capabilities by monitoring world-readable files and presenting the user with a contextual notification. An attacker can mount the attack assuming that she controls an untrusted app with no permissions, as we showed with our malicious app.

8. Mitigations

Even if we have shown that our attack also works in case the attacker cannot monitor files in the installation path, this capability is the cornerstone of our work that Google has acknowledged. Therefore, it is evident that the primary fix is to avoid untrusted apps from watching files in the installation path, even on their own, because we have also noticed malware abusing it to execute code when uninstalled. To propose a solution to this vulnerability, it is imperative to understand the root cause that allows an attacker to exploit it. Given that Android uses SELinux to enforce MAC which uses a "default-deny" approach, as explained in Section 2, there must be rules that explicitly allow monitoring of the installation folder.

Watching the installation path. As a first step, we inspected the file where SELinux kernel hooks are defined to determine whether there was any filtering related to inotify syscalls [17]. We identified that SELinux sets up the `selinux_path_notify` hook for handling these syscalls. In particular, the hook verifies if the context of the subject contains the `watch` and `watch_reads` SELinux permissions. The difference between the two lies in the fact that `watch_reads` is only related to read-like events.

Then, we proceeded to check if any SELinux rules would allow a third-party app (running in the context of `untrusted_app`) to use inotify (i.e., allowing `watch` or `watch_reads`) on files or subfolders of the `/data/app/` (which are labeled as `apk_data_file`). It is worth noting that, on Android, the `untrusted_app` context inherits all the `appdomain` rules. After retrieving the compiled SELinux policies from our emulator (namely, the `/sys/fs/selinux/policy` file), we searched for these rules, and our hunch turned out to be correct. We found two rules that allow the `appdomain` (thus, `untrusted_app`) to use both `watch` and `watch_reads` on files and directories labeled as `apk_data_file` (i.e., files and directories in `/data/app`). A deeper analysis allowed us to identify how both `watch` and `watch_reads` permissions are defined in the `r_file_perms` and `r_dir_perms` macros [20].

As mitigation for this vulnerability, acting at the SELinux level is the best choice as it ensures a strong level of security and controls its granularity simultaneously. For example, removing the POSIX world-readable permission on the `base.apk` would be wrong. A typical use case is antivirus software, which should be able to read this file for its analysis. Therefore, we propose to create a new macro (e.g., `r_file_perms_nowatch`) with the same permissions of `r_file_perms` but without `watch` and `watch_reads`. Then, this new macro can be used to define the permissions of `appdomain`, and thus consequently `untrusted_app`, for operating on `apk_data_file`. This way, we can prevent an untrusted app from listening for inotify events on files or directories labeled `apk_data_file` without limiting access to this file. However, we can always allow third-party apps to use inotify, for instance, on files or directories belonging to their sandbox, which are legitimate uses and scenarios. This mitigation effectively eliminates the main vulnerability we found while not compromising the stability and usability of the system and introducing a highly fine-grained level of control on which processes can use inotify and where.

We implemented this mitigation by modifying the AOSP source, and our emulator works correctly but without allowing monitoring files within the installation path with inotify. However, only Google has the resources to test if this change negatively impacts the whole ecosystem.

RQ5 – Third-party vendors. Since Possemato et al. [37] have shown a significant difference in the security posture of AOSP w.r.t. Original Equipment Manufacturers, we investigated if their flagship smartphones allow using inotify on the installation path. We collected the major vendors' latest ROMs (Samsung, Xiaomi, Motorola, Lenovo, and Realme – Android 12), resulting in a dataset of 11 images. Then, we extracted the content of each ROM, looking for the SElinux policy files. However, the `/sys/fs/selinux/policy` file is available only at runtime; thus, we searched for the source file (that changes depending on the customization). Finally, *all* the

tested ROMs inherit the default AOSP SELinux policy, which is also vulnerable. We provide more technical details and the list of the ROMs in Appendix F.

Information disclosure. Then, the second most common paths found in the $FSSignature^{*}_{K}$ (Section 6) and the information disclosure vulnerability (Section 7.3) are related to the external storage (i.e., `/storage/`). These vulnerabilities are easily solvable with POSIX permissions. All files and folders in the private external storage should be read-write-execut-able by the owner. At the same time, to solve the information disclosure, the `Android/data` folder in the emulated storage must not be world-executable. Moreover, to err on the side of caution, it would also be appropriate to introduce a random string in the presence of folders with the package name of the app (e.g., `/storage/emulated/0/Android/data/com.discord/`), as is already the case in the installation path.

RQ6. There are mitigations, and Android system developers should act in combination with SELinux and POSIX permissions. SELinux policies should be extended to prevent an untrusted app from being able to install inotify watches in any installation path, and just the owner should be able to operate on her private files in the external storage. Moreover, similarly to the installation path, folders named with the package name of an installed app can lead to information disclosure or path traversal; therefore, they should be accompanied by a random string. However, we argue that comprehensive remediation does not exist and Attack #2 can always be attempted. As in the Man-in-the-Disk, the $FSSignature_K$ of an app depends on its specific behavior in combination with one of the other installed apps, thus resulting in a complex environment that is difficult to defend.

9. Limitations & Conclusions

Limitations. This work is not exempt from limitations.

First, the timespan considered as the startup time of the app crucially determines the events in the $FSSignature_K$, and, thus, the attack's impact. If the timespan is too short, potential events are lost, limiting the attack; if it is too long, the app may have already started when the attack takes place, and the user may notice odd behavior.

Second, we collected the file system events stimulating each app with ARES. Thus, we inherit the limitations of the dynamic analysis [2], [36]: such traces may not be complete because the analysis did not lead to specific code.

Third, some apps' components (e.g., ads libraries) may show different behaviors depending on the device on which they are executed (e.g., actual device vs. emulator). In the case of advertisement libraries, they are part of a shared codebase that would likely not generate unique events. These types of evasive checks are usually a phenomenon typical of malware.

Fourth, in our analysis, we dwelt on monitoring a single file, and given its effectiveness, we explored no further; Ahn et al. [3], instead, focused on a sequence of events. Although there is a margin for improvement, our work is mainly concerned with showing the existence of this new attack and how to stop it. Future works must consider the actions Google will take after our disclosures.

Google Play Store. We have always referenced Google Play Store's policies and how they may limit our attack. In particular, we emphasized that if an app that requires the `QUERY_ALL_PACKAGES` permission (thus, looking for the installation path) is uploaded to the Play Store, the app's use of this permission is subject to approval based on specific security policies. Even assuming Google fixes `queries` tag workaround, the level of such permission is "normal" (i.e., automatically granted at installation time); therefore, there are two crucial factors to consider.

First, all the restrictions imposed by Google on its store have led malware authors not to upload their malicious apps directly. They upload droppers to side-load the actual malicious app with extended permissions [34], [48], [51]. Therefore, in such a scenario, our attack has no limitations.

Second, the Play Store is one of the many app distribution channels. For instance, in China, the Google Play market share is less than 4%, while MyApp (Tencent) [50] currently holds 25% of the market. Therefore, such a plethora of alternative markets (e.g., Samsung Galaxy App [43], Amazon App Store [5], AppBrain [7], etc.) imply that our attack has a tremendous impact on the global Android ecosystem.

Closing remarks. We introduced the concepts of file system footprint and signature and showed how these "traces" lead to figuring out when a particular app is starting. Then, thanks to inotify, we can execute code promptly to mount a phishing attack – this work also opens up interesting future works because these concepts can be ported to different operating systems. Our measurements show that *all* Android apps are vulnerable if the attacker can monitor a file in the installation path. Otherwise, we have shown that it is still possible to carry it out on a smaller but significant number of apps. What is more, the attacker can unleash it from a malicious app without requesting any permission. Fortunately, we have also shown the existence of practical remediation. However, even assuming they are all implemented immediately, the number of devices left vulnerable will remain high for many years because of the timeframe to deploy these updates, and many device vendors will not apply them.

In conclusion, despite the implications of our attack, it has so far remained unknown to malware authors. We hope they got scooped and that our contribution came in time to prevent many personal data thefts.

Responsible Disclosure. We have reported three vulnerabilities to Google through the official issue tracker. I) The possibility of using inotify in the installation folder (reported in May 2022, acknowledged the same month); II) The information disclosure, described in Section 7.3 (reported in July and acknowledged in September 2022); III) The package visibility bypass via the `queries` tag, described in Section 7.1 (reported in July 2022). The first two were acknowledged as bugs of Moderate severity [28], while the third was previously reported by other researchers but was not public.

Acknowledgements. This work has benefited from a government grant managed by the National Research Agency under France 2030 with the reference "ANR-22-PECY-0007." This work was also partially supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

References

[1] N. N. S. Agency, "Ghidra: A software reverse engineering (sre)," https://ghidra-sre.org/, 2022, accessed March 1, 2023.

[2] A. Aggarwal and P. Jalote, "Integrating static and dynamic analysis for detecting vulnerabilities," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, vol. 1. IEEE, 2006, pp. 343–350.

[3] W. H. Ahn, S. Park, J. Oh, and S.-H. Lim, "Inishing: a ui phishing attack to exploit the vulnerability of inotify in android smartphones," *IEICE TRANSACTIONS on Information and Systems*, vol. 99, no. 9, pp. 2404–2409, 2016.

[4] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 468–471.

[5] Amazon, "Amazon app store," https://www.amazon.com/gp/mas/get/amazonapp, 2022, accessed March 1, 2023.

[6] S. Aonzo, A. Merlo, G. Tavella, and Y. Fratantonio, "Phishing attacks on modern android," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1788–1801.

[7] AppBrain, "App brain," https://www.appbrain.com/, 2022, accessed March 1, 2023.

[8] AppMetrica, "Appmetrica yadex," https://appmetrica.yandex, 2022, accessed March 1, 2023.

[9] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, "What the app is that? deception and countermeasures in the android user interface," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 931–948.

[10] D. Caputo, L. Verderame, S. Aonzo, and A. Merlo, "Droids in disarray: detecting frame confusion in hybrid android apps," in *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2019, pp. 121–139.

[11] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it:{UI} state inference and novel android attacks," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 1037–1052.

[12] T. M. Corporation, "Software discovery," https://attack.mitre.org/techniques/T1418/, 2022, accessed March 1, 2023.

[13] A. P. Felt and D. Wagner, "Phishing on mobile devices," 2011.

[14] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash, "Android ui deception revisited: Attacks and defenses," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 41–59.

[15] F. File, "Firmware file," firmwarefile.com, accessed March 1, 2023.

[16] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and dagger: from two permissions to complete control of the ui feedback loop," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 1041–1057.

[17] Google, "Android 12.1 – selinux hooks," https://android.googlesource.com/kernel/common/+/refs/tags/android-12.1.0_r0.35/security/selinux/hooks.c, 2021, accessed March 1, 2023.

[18] ——, "Android 8.0 – security changes," https://developer.android.com/about/versions/oreo/android-8.0-changes#security-all, 2021, accessed March 1, 2023.

[19] ——, "Android app categories," https://support.google.com/googleplay/android-developer/answer/9859673, 2022, accessed March 1, 2023.

[20] ——, "Android mainline – selinux macro," https://android.googlesource.com/platform/system/sepolicy/+/refs/heads/master/public/te_macros, 2022, accessed March 1, 2023.

[21] ——, "Android package visibility," https://developer.android.com/training/package-visibility, 2022, accessed March 1, 2023.

[22] ——, "Android permissions," https://developer.android.com/guide/topics/permissions/overview, 2022, accessed March 1, 2023.

[23] ——, "Android scoped storage," https://developer.android.com/training/data-storage#scoped-storage, 2022, accessed March 1, 2023.

[24] ——, "Average number of apps installed on users' smartphones," https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/average-number-of-apps-on-smartphones/, 2022, accessed March 1, 2023.

[25] ——, "Create a custom notification layout," https://developer.android.com/training/notify-user/custom-notification, 2022, accessed March 1, 2023.

[26] ——, "Declaring package visibility needs," https://developer.android.com/training/package-visibility/declaring, 2022, accessed March 1, 2023.

[27] ——, "Fileobserver," https://developer.android.com/reference/android/os/FileObserver, 2022, accessed March 1, 2023.

[28] ——, "Security updates and resources – severity," https://source.android.com/security/overview/updates-resources#severity, 2022, accessed March 1, 2023.

[29] T. kernel development community, "Bpf documentation — the linux kernel documentation," https://www.kernel.org/doc/html/latest/bpf/index.html, 2022, accessed March 1, 2023.

[30] S. Kevin, "Bankbot found on google play and targets ten new uae banking apps." https://www.trendmicro.com/en_us/research/17/i/bankbot-found-google-play-targets-ten-new-uae-banking-apps.html, accessed March 1, 2023.

[31] R. Love, "inotify documentation," https://www.kernel.org/doc/Documentation/filesystems/inotify.txt, 2015, accessed March 1, 2023.

[32] C. P. S. T. Ltd., "Man-in-the-disk: A new attack surface for android apps," https://blog.checkpoint.com/2018/08/12/man-in-the-disk-a-new-attack-surface-for-android-apps/, 2018, accessed March 1, 2023.

[33] L. Malisa, K. Kostiainen, and S. Capkun, "Detecting mobile application spoofing attacks by leveraging user visual similarity perception," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 289–300.

[34] Malwarebytes, "Trojan.dropper," https://blog.malwarebytes.com/detections/trojan-dropper/, 2022, accessed March 1, 2023.

[35] F. Naves, A. Conway, S. W. Jones, and A. Mcneil, "Tanglebot: New advanced sms malware targets mobile users across u.s. and canada with covid-19 lures," https://www.cloudmark.com/en/blog/malware/tanglebot-new-advanced-sms-malware-targets-mobile-users-across-us-and-canada-covid-19, accessed March 1, 2023.

[36] A. Petukhov and D. Kozlov, "Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing," *Computing Systems Lab, Department of Computer Science, Moscow State University*, pp. 1–120, 2008.

[37] A. Possemato, S. Aonzo, D. Balzarotti, and Y. Fratantonio, "Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2021.

[38] A. Possemato, D. Nisi, and Y. Fratantonio, "Preventing and detecting state inference attacks on android," in *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS), Virtual, 21st-25th February*, 2021.

[39] C. Ren, P. Liu, and S. Zhu, "Windowguard: Systematic protection of gui security in android." in *NDSS*, 2017.

[40] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu, "Towards discovering and understanding task hijacking in android," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 945–959.

[41] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep reinforcement learning for black-box testing of android apps," *ACM Transactions on Software Engineering and Methodology*, 2022.

[42] A. Ruggia, "inotify-analyzer," https://gitlab.eurecom.fr/totoR13/inotify-analyzer, 2023, accessed March 1, 2023.

[43] Samsung, "Samsung galaxy app," https://www.samsung.com/levant/apps/galaxy-store/, 2022, accessed March 1, 2023.

[44] S. Sebastián and J. Caballero, "Avclass2: Massive malware tag extraction from av labels," in *Annual Computer Security Applications Conference*, 2020, pp. 42–53.

[45] A. Security, "libbpfgo," https://github.com/aquasecurity/libbpfgo, 2022, accessed March 1, 2023.

[46] R. Spreitzer, F. Kirchengast, D. Gruss, and S. Mangard, "Procharvester: Fully automated analysis of procfs side-channel leaks on android," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 749–763.

[47] R. Spreitzer, G. Palfinger, and S. Mangard, "Scandroid: Automated side-channel analysis of android apis," in *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2018, pp. 224–235.

[48] G. Stergiopoulos, D. Gritzalis, E. Vasilellis, and A. Anagnostopoulou, "Dropping malware through sound injection: A comparative analysis on android operating systems," *Computers & Security*, vol. 105, p. 102228, 2021.

[49] stockrom.net, "stockrom," stockrom.net, accessed March 1, 2023.

[50] Tencent, "Myapp – tencent," https://android.myapp.com/, 2022, accessed March 1, 2023.

[51] ThreatFabric, "300.000+ infections via droppers on google play store," https://threatfabric.com/blogs/deceive-the-heavens-to-cross-the-sea.html, 2021, accessed March 1, 2023.

[52] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[53] Vungle, "Vungle," https://support.vungle.com/hc/en-us/, 2022, accessed March 1, 2023.

[54] D. Web, "The coper — a new android banking trojan targeting colombian users," https://news.drweb.com/show/?i=14259&lng=en&c=5, accessed March 1, 2023.

[55] L. Wu, X. Du, and J. Wu, "Mobifish: A lightweight anti-phishing scheme for mobile phones," in *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2014, pp. 1–8.

[56] Z. Xu and S. Zhu, "Abusing notification services on smartphones for phishing and spamming." in *WOOT*, 2012, pp. 1–11.

[57] C. Youmeng, "Umeng," https://www.umeng.com/, 2022, accessed March 1, 2023.

[58] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang, "Leave me alone: App-level protection against runtime information gathering on android," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 915–930.

A. Distribution of malware families

Table 2 reports the five most frequent families and the Singletons for the whole malware dataset (row 2), malware that interacts with inotify in Java (row 3), and native layers (row 4).

B. Inotify Events to System Calls

As described in Section 5.2, *inoTool* creates the *FSFootprint* from the list of system calls that eBPF intercepted. While, in most cases, mapping a syscall to an inotify event is straightforward (such mapping is reported in Table 3), three event types represent interesting exceptions: file closing, moving, and deleting. The first event type occurs when a process closes a file descriptor and distinguishes two different cases. If the process had opened the file descriptor in writing mode, then an (`IN_CLOSE_WRITE`) event would arise; on the contrary, inotify returns an (`IN_CLOSE_NOWRITE`) event if the process did not open the file descriptor for writing. To correctly handle these two events, *inoTool* keeps track of the opening options for each file descriptor. The *move* event class distinguishes three different events. The first one – `IN_MOVE_SELF` – happens when the moved file is the program that spawned the process that moves the file. The other two – `IN_MOVE_TO` and `IN_MOVE_FROM` – arise when the file is moved *to* or *from* the monitored directory, respectively. Similarly, the *delete* event class depends on whether the watched file/directory was itself deleted (`IN_DELETE_SELF`) or the file/directory deleted from the watched directory (`IN_DELETE`).

C. App-dependent events path

Table 4 presents the distribution of *app-dependent* events of our dataset throughout an Android file system. Specifically, each row in the table represents a path in the Android file system, defined in terms of top- and second-level (first and second column, respectively) directories. The third column provides the share of apps in the dataset whose $FSSignature^{*}_{K}$ contain at least one event in the corresponding directory.

D. Android APIs & private folder

During our analysis, we investigated which Android APIs are used to to retrieve the path(s) of the private folder in the external memory. In Table 5, we show the prevalence in our dataset of benign apps (Section 7). While in Table 6, we report all the "tainted" APIs we found in malware (Section 4), that is, the ones involved in generating the path passed as input to the `FileObserver` class.

E. SD cards

In Android, handling external memories (especially SD cards) has changed a lot between versions. In the past, `/sdcard/` folder was used to this purpose, while today it is a symbolic link to the `/storage/self/primary/` folder, which in turn is always (counterintuitively) a symbolic link to `/storage/emulated/0/`. However, regardless of the presence or absence of a physical

Table 2: Distribution of the malware families

Dataset	# of Samples	# of Families	Fam1 (%)	Fam2 (%)	Fam3 (%)	Fam4 (%)	Fam5 (%)	Singleton (%)
	10,000	140	5.0	0.8	0.8	0.7	0.5	84.0
Java layer	744	65	4.8	3.9	2.4	2.1	2.0	57.8
Native layer	769	30	0.23	0.19	0.12	0.09	0.08	98.7

Table 3: Inotify event to system call

Inotify event	System call
IN_ACCESS	read execve execveat
IN_ATTRIB	chmod fchmod fchmodat utimensat setxattr lsetxattr fsetxattr chown fchown lchown fchownat utime utimes futimesat
IN_CLOSE_(WRITE|NOWRITE)	close
IN_CREATE	mkdir mkdirat link linkat symlink symlinkat bind mknod mknodat
IN_DELETE(_SELF)	rmdir unlink
IN_MODIFY	write truncate ftruncate
IN_OPEN	open openat
IN_MOVE_(SELF|FROM|TO)	rename renameat renameat2

Table 4: Path distribution of files in $FSSignature^*_K$

Root	Depth=1	Percentage %
/data		17.20
	/app	17.20
/storage		1.5
	<sdcard_ID>	0.68
	/emulated	0.84
/sys		0.06
	/devices	0.06
/system		0.03
	/priv-app	0.03
	/lib64	0.02
/etc		0.02
	/passwd	0.02
/product		0.02
	/overlay	0.02

Table 5: Android APIs usage to interact with the external memory. We omitted `android` at the beginning of each API for readability.

Path	%
.content.Context.getFilesDir	98.73
.os.Environment.getExternalStorageDirectory	97.15
.content.Context.getExternalFilesDir	91.35
.content.Context.getExternalFilesDirs	90.34
.content.Context.getExternalCacheDirs	90.06
.content.Context.getExternalCacheDir	88.10
.content.Context.getExternalMediaDirs	87.07
.content.Context.getObbDirs	74.73
.content.Context.getObbDir	61.96

Table 6: Tainted Android APIs that affects `FileObserver` constructor in malware

Path	%
android.content.Context.getFilesDir	11.32
android.content.Context.getNoBackupFilesDir	7.84
android.content.Context.getExternalFilesDir	4.35
android.content.Context.getDir	1.74
android.os.Environment.getExternalStorageDirectory	0.44

external mass memory device (i.e., SD card), the Android system always "emulates" an external device under the folder `/storage/emulated/`, where it mounts a partition of its internal memory.

Nowadays, when a user inserts an SD card, the system offers her two configurations. The first, named *portable*, manages the memory space on the SD card so the user can remove and use it on other devices. When a new SD card is plugged in, the system creates the mount point `/storage/<sdcard_ID>/`. Instead, the *internal* configuration is designed to extend the device's internal memory. What happens under the hood is that the system mounts the SD card in the emulated external storage, i.e., under `/storage/emulated/`. In this scenario, the SD card cannot be transferred between different devices.

During our analysis, we discovered that apps' private folders in the *portable* configuration of an SD card could be monitored by means of the inotify API (contrary to what an attacker is able to do in the internal memory).

F. Third-party vendors

We collected other vendors' latest ROMs from `stock-rom.net` [49] or `firmwarefile.com` [15] for the most popular vendors – i.e., Samsung, Xiaomi, Motorola, Lenovo, and Realme (see Table 7). To gather reliable results, we collected only the ROMs with the latest security upgrades (e.g., scoped storage), preferring Android 12 when available. Then, we extracted the content of each rom, looking for the SElinux policy files. It is worth noticing that the `/sys/fs/selinux/policy` file is available only at runtime; thus, we searched for the following source file (it could depend from the ROM customization): `sepolicy`, `file_contexts`, `property_contexts`, `seapp_contexts`, `service_contexts`, `plat_file_contexts`, `nonplat_file_contexts`, `plat_property_contexts`, `nonplat_property_contexts`,

Table 7: ROM versions & models

Brand	Model	Vulnerable
Samsung	Galaxy S20 Ulta	✓
	Galaxy S10e	✓
	Galaxy A23	✓
Xiaomi	Xiaomi 12S Pro	✓
	Redmi Note 8	✓
	Redmi 10	✓
Motorola	Edge 20 Pro	✓
	Moto G52	✓
	Moto G30	✓
Realme	C25 RMX3191	✓
Lenovo	Legion Y90	✓

`plat_service_contexts`, `nonplat_service_contexts`, `plat_hwservice_contexts`, `nonplat_hwservice_contexts`, `vndservice_contexts`, `plat_seapp_contexts`, `nonplat_seapp_contexts`.

The analysis allowed us to identify that all the tested ROMs inherit the AOSP SELinux policy, which assigns to an untrusted app the `watch` and `watch_read` permissions (refer to Section 8).