Preventing and Detecting State Inference Attacks on Android

Andrea Possemato
IDEMIA and EURECOM
andrea.possemato@eurecom.fr

Dario Nisi
EURECOM
dario.nisi@eurecom.fr

Yanick Fratantonio
EURECOM and Cisco Talos
yanick@fratantonio.me

***Abstract*—In the realm of the Android ecosystem, one relevant threat is posed by phishing attacks. Phishing attacks are particularly problematic for mobile platforms because they do not provide enough information for a user to reliably distinguish a legitimate app from a malicious app spoofing the UI of the legitimate one. A key factor that determines the success rate of a phishing attack is proper timing: The user is more prone to provide sensitive data (such as her passwords) if the malicious spoofed UI appears when the victim expects to interact with the target app. On Android, malware determines the right timing by mounting so-called *state inference attacks*, which can be used, for example, to infer the exact moment that the user started a target app and thus expects to interact with it. Even though Android app sandbox is designed to prevent these attacks, they are still possible by abusing vulnerable APIs that leak such sensitive information: the usual scenario is a malicious app that "polls" these vulnerable APIs, infers when a target app is about to be used by the user, and makes the spoofed UI appear on top of the screen at the right time. All previous bugs of this kind have been fixed in the latest version of Android.**

This paper presents two main research contributions related to preventing and detecting state inference attacks. First, we discuss the design and implementation of a new vulnerability detection system, which specifically aims at identifying *new* vulnerabilities that can be used to mount state inference attacks. Our approach relies on both static and dynamic analysis techniques and it identified 18 previously unknown bugs (leading to 6 CVE) in the latest versions of Android. Second, we present a new on-device analysis system able to detect exploitation attempts of vulnerable resources and APIs. This system is based on the key hypothesis that mere "polling behaviors" can be used as a strong signal of a potential attack, independently of other factors (that previous works rely on). We performed an empirical analysis over a corpus of benign and malicious apps, and we find that this hypothesis is indeed correct. This approach has the advantage of being able to detect exploitation attempts even when the abused API is *not* known to be vulnerable in advance. We implemented this system as an Android framework modification, and we show it incurs a negligible overhead.

I. Introduction

One of the key security features of Android is the application sandbox. This mechanism aims at enforcing a strong security boundary between different apps and protects sensitive information. One of such sensitive information is the "state" a given app is currently in. With "state," we refer to, for example, whether an app is currently in the background, in the foreground, or is transitioning between these states. Attacks aiming at determining the state of another app are called *state inference attacks*, which are particularly relevant in the context of phishing attacks. Phishing attacks consist of luring an unsuspecting user into revealing her sensitive information (e.g., credentials) to a malicious app that mimics the UI of the legitimate one, a technique we refer to as *UI Spoofing*. The peculiar problem of mobile platforms is that the user cannot understand whether she is inserting her credentials into a legitimate app or into a malicious app spoofing its UI. State inference attacks play a key role in this context since, if the malicious app can infer, for example, that *the user is about to use a specific app*, it can show the spoofed UI at the *right time*, and hijack the legitimate app's flow.

In the context of Android security, malicious apps are able to leak this state-related information by exploiting vulnerable APIs or resources (e.g., `/proc` file system). For example, a vulnerable API, when invoked with specific arguments, may return data that can be used to infer whether another app was just started. These attacks have been known for several years, and previous works have shown that several APIs and resources do leak sensitive information [6], [4]. Given the security implications of these vulnerabilities, Google has restricted access to the `/proc` file-system (eradicating potential bugs at its root) and fixed all APIs known to be vulnerable [18]. However, as for many forms of bugs, this is an arms race and there can potentially be many more vulnerable APIs still undisclosed.

As the first contribution of this paper, we design, implement, and evaluate a new analysis framework to automatically pinpoint Android APIs that may disclose state-related information about other apps or the operating system itself. The main idea is to first systematically enumerate the attack surface in terms of which APIs could be potentially abused, to then repeatedly invoke each API (with appropriate arguments) while changing the surrounding context (e.g., another app is started), and finally, to monitor how the returned values change (if they do) depending on such context. We note that we are not the first ones to propose this research direction. A recent work that tackles a similar problem is SCAnDroid [27], which attempts to employ a technique similar to ours. Our paper shows that while SCAnDroid's direction is indeed promising, there are several conceptual and technical challenges that were overlooked, leading to undetected vulnerabilities. One of the main problems we uncover is that previous works have mischaracterized the attack surface available to a malicious app, leading to many APIs to not be even selected as candidates for analysis in the first place: our analysis shows that it considered only $\sim$44% of the attack surface. One other open challenge is *how* each of these APIs should be analyzed to uncover potential problems, and previous works oversimplified this step as well. As the last example, we found that even the task of determining whether the return value of an API contains sensitive information can

Network and Distributed Systems Security (NDSS) Symposium 2021
21-24 February 2021, San Diego, CA, USA
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24479
www.ndss-symposium.org

be challenging, and we find that this is another venue for mistakes.

Our paper systematizes these challenges, discusses how we address them, and shows that *each of these overlooked challenges is the direct cause of false negatives of the closest related work, SCAnDroid* (see Section VI-F for a detailed comparison). We tested the effectiveness of our framework on Android 8.1, 9.0, and 10, unveiling 18 previously unknown bugs. All the vulnerabilities were reported to Google and several of these have been already acknowledged and fixed.

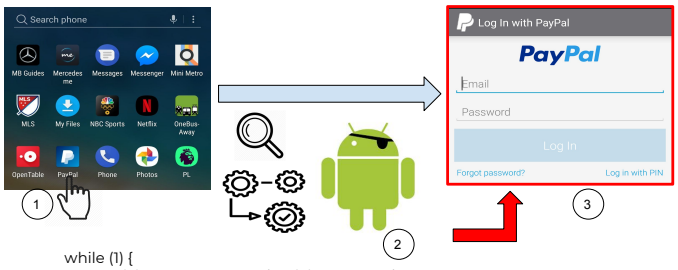
While we believe that our framework is a good first step to automatically detect this category of bugs, we acknowledge that identifying and removing *all* vulnerable APIs is not always possible. Thus, as a second contribution of this paper, we design and implement an on-device monitoring system to detect state inference attacks when they occur. This system builds on two observations: 1) *all* existing state inference attacks implement a *polling behavior*, thus making it per-se a good candidate for detection; 2) the second observation, which, to the best of our knowledge, has not been explored before, is based on the following key hypothesis: *benign apps rarely rely on polling and, when they do, the nature of their behaviors is different than those of malicious apps.* In other words, *if benign apps do not commonly employ polling*, the mere detection of these behaviors could be then used as a strong signal for flagging an app as suspicious.

To verify the validity of our hypothesis, we performed an empirical study over *all* known families of malware exploiting vulnerabilities to perform phishing attacks, as well as on a set of more than 10,000 popular benign applications. The results of this experiment show that, as expected, all malicious samples implement some form of polling when mounting state inference attacks. For what concerns the benign apps, our study unveils a surprising insight: there are several benign apps that also perform polling; However, more in-depth experiments show that these behaviors are of different natures, and it is easy to distinguish between them and their malicious counterparts. We thus show that polling itself can be leveraged as a strong signal to detect state inference attacks. We implemented this system as a modification to the Android framework, and our experiments show that this system would incur a negligible overhead.

We note that using "polling detection" as a mean to identify malicious apps is not novel per-se: a previous work, LeaveMeAlone [38], has explored this aspect. However, we show how this related work is not suitable when tasked to detect phishing attacks on modern versions of Android. We offer a detailed comparison in Section IX-E. We thus believe that our work discusses a new interesting point in the design space of detection approaches.

In summary, this paper makes the following contributions:

- We systematize and pinpoint open challenges to tackle the automatic detection of APIs vulnerable to state inference attacks. Among these, we show that the attack surface is bigger than what previously thought.
- We implement an automatic framework to unveil vulnerable APIs leading to state inference attacks. We tested its efficacy on Android 8.1, 9.0, and 10, identifying 18 new vulnerable APIs (and 6 CVE were assigned).
- We hypothesize that the mere polling can be used as a strong signal to identify in-progress state inference attacks. To validate our hypothesis, we performed an empirical study on both malware and benign applications, and we show it is indeed possible to reliably and efficiently pinpoint attacks. This can be the basis for an on-device detection system that does not have the limitations affecting previous works.




```
while (1) {
    ActivityManager am = (ActivityManager)
    Context.getSystemService(ACTIVITY_SERVICE);
    List<ActivityManager.RunningTaskInfo> apps = am.getRunningTasks(1);
    String app = apps.get(0).topActivity.getPackageName();
    if (app.equalsIgnoreCase(TARGET_APP)) {
      // Perform the attack
    }
}
```

Fig. 1: Anatomy of a phishing attack.

In the spirit of open science, we will release all the source code developed for this paper and the relevant datasets.

II. Phishing Attacks on Android

This section provides the technical background about phishing attack and how a malicious application can successfully mount it. It then discusses a systematic survey on all known classes of state inference vulnerabilities, their role in the context of phishing attacks, and which of these classes are still problematic on Android.

A. Phishing

One common task of Android malware is "phishing." With this term, we refer to malicious applications trying to steal user's sensitive information (e.g., credentials). Phishing attacks are particularly problematic for mobile platforms because they do not provide enough information for a user to reliably distinguish a legitimate app from a malicious app spoofing its UI. To make the attack more effective, malware relies on the ability to mount state inference attacks, useful to monitor when the user is about to interact with a target app. Inferring the right time is important, as it allows a malicious app to ask for user's credentials exactly when the user expects to insert them. We note that these techniques are not only known and studied in the academic world [12], [6], [23], [26], [27], but they are used by real-world malware [15], [28], [32], [21], [17], [20], [29], [16], [31].

B. Anatomy of a Phishing Attack

Figure 1 depicts the various phases of a phishing attack. We start from a scenario where the user (①) wants to interact with a sensitive app (e.g., PayPal). Meanwhile, in the background, the attacker (②) repeatedly invokes the once-vulnerable API `getRunningTasks` API to determine which app is in foreground. Before the victim clicks on the PayPal icon, the attacker could determine that the foreground app is the "Home Launcher." However, by repeatedly invoking this API and checking its return value, the attacker could mount a state inference attack and infer the exact moment the user clicks on the PayPal icon: the attacker would in fact notice the transition from the Home Launcher to the PayPal app. At this point, the attacker knows

this is the best time to hijack the PayPal activity with a spoofed one which looks the same as the original (③). A successful attack will leave the user completely unsuspecting since the victim initiated the interaction with the target app herself, she would not find an authentication request from that particular target app unexpected.

C. Characterizing State Inference Attacks

Previous research identified many venues to mount state inference attacks [12], [6], [23], [26], [27]. With the goal of better characterizing this threat and to better understand the state-of-the-art of Android state inference attacks, we analyzed all the different vulnerabilities exploited by malware and discovered during the years [12], [6], [23], [26], [27], [15], [28], [32], [21], [17], [20], [29], [16], [31]. All existing vulnerabilities can be grouped into two conceptual categories:

Filesystem layer. The first category relates to the *filesystem* layer. The root cause of these vulnerabilities resides in the presence of sensitive information obtainable by reading files accessible by any unprivileged app. From the technical standpoint, all known vulnerabilities are caused by unrestricted access to `procfs`, via the `/proc` directory. For example, one of the first vulnerabilities relied on accessing `/proc/$PID/cmdline`, which contains the name of the program run by a process with a given `$PID`. By continuously monitoring the content of this directory, the attacker could identify the creation of new processes (by monitoring sub-directories of `/proc`), and infer the app started by the user (by reading the `cmdline` file).

Many similar vulnerabilities were discovered, but they all had the same root cause: unprivileged apps had access to `procfs`. Thus, to patch these vulnerabilities, from Android 7.0 the access to almost the entire `/proc` directory is forbidden. We believe this solution eradicates this category of vulnerabilities at its root.

Android System Services layer. The second category of vulnerabilities relates to *Android System Services*. Services are a fundamental sub-system in the Android Framework. They allow apps to interact with "lower" operating system and hardware components, such as GPS, network, etc. Since this operation normally requires interaction with privileged components, services are offered by a process called `system_server`, which runs as the privileged `system` user. This process is in charge of handling almost all the core services and provides a bridge between the functionality requested by the app and the service implementing it.

We note that *all* API-related vulnerabilities identified by previous works relate to APIs exposed by services. Even though Google has fixed all known vulnerabilities, the complexity of the services infrastructure makes it significantly more challenging to identify a single root cause that led to all existing vulnerabilities. Moreover, we show how there are several previously overlooked challenges and subtleties that make the automatic vulnerability discovery process more difficult than what previously thought, and that this is the direct cause for false negatives in recent related works [27].

III. Threat model

We consider a threat model in which an attacker controls a malicious app on the victim's phone. We also assume that such app can ask (and obtain) those permissions that are usually available to non-system third-party apps. Some of these permissions are automatically granted, while others require user interaction.

An example of a permission automatically granted is the `INTERNET` permission: at installation time, the system grants this permission to the application and no user interaction is required.

Instead, examples of permissions that require user interactions to be granted are `ACCESS_COARSE_LOCATION` or `PACKAGE_USAGE_STATS`. Note that, in Android, this *interaction* may be implemented is two ways.

The first type of interaction relies on *runtime prompt* and it is used to grant the permissions labeled as *dangerous*, like the `ACCESS_COARSE_LOCATION` permission. By interacting with this prompt, the user can decide whether to grant or deny the permission to the app.

The second type of interaction, which does not rely on prompts, is reserved for privileged permissions. These permissions might be labeled as *signature, system, signatureOrSystem, privileged, development, appop, or retailDemo*. An example of this category of permission are the `PACKAGE_USAGE_STATS`, `SYSTEM_ALERT_WINDOW`, and `BIND_NOTIFICATION_LISTENER_SERVICE` permissions. For example, the `PACKAGE_USAGE_STATS` permission is used to mainly protect the *UsageStatsManager* service [13]. With that being said, Android offers a mechanism for third-party apps to obtain sensitive information accessible only via these permissions, even without technically being granted such permissions. The way it works is that a third-party application can ask the user of the device to grant the permission through the System Settings app, which updates some internal settings. The sensitive system services that do have the signature permissions then check such settings to determine whether a requesting app is entitled to have access to such sensitive permission-protected information. We also note that not only is it possible to access information protected by these signature-level permissions, but that many real-world apps (both benign and malicious) currently use them [20], [35], [5]. Thus, since third-party applications may require some of these permissions, we believe it is appropriate to consider them within our threat model.

We also assume the malicious app *cannot* obtain the `BIND_ACCESSIBILITY_SERVICE` permission (`a11y`): this permission alone allows an attacker to fully monitor all UI events [12], making mounting phishing attacks trivial. Finally, we do *not* consider the scenario where a malicious app can gain root privileges: once again, these powerful attackers can easily steal sensitive information without mounting phishing attacks.

IV. Exploring the Attack Surface: System Services

This paper aims at developing an automated approach to identify vulnerable APIs that could be used to mount state inference attacks. For the aforementioned reasons, we focus on considering the attack surface exposed by *System Services*. This section discusses the inner workings of system services and the known security-related pitfalls.

A. Android System Services

System Services are the key mechanisms for apps to interact with low-level, security-sensitive operating system and hardware components. The technical details of these mechanisms, and how third-party apps can rely on them (by means of invoking Android APIs) are not trivial, and it involves several sub-components, discussed next.

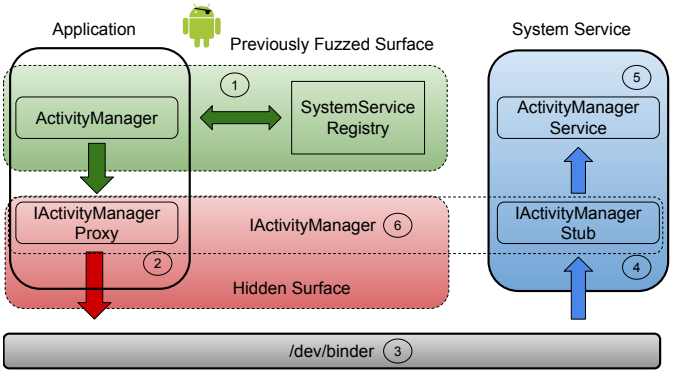


Fig. 2: Interaction with ActivityManagerService, showing the client-server structure of Android System Services.

Figure 2 gives an overview of how system services work. In the example, the goal of an app is to interact with the ActivityManagerService: to do so, it needs to first request a "client" — named `Manager` (①) — to the `SystemServiceRegistry` class to interact with the service. Once obtained, it can start invoking the methods exposed by the Manager. Each method invocation is then wrapped and forwarded to another component, named `Proxy` (②), in charge of sending the data from the application to the `Binder` component (③). This component "forwards" the request to its associated `Stub` (④), which can be seen as the counterpart of the Proxy, residing in the Service. Finally, Stub forwards the request to the actual implementation of the `Service` (⑤). The response follows the same, but reversed, flow. Another important technical aspect is represented by the Interface (⑥), written in AIDL (Android Interface Definition Language). AIDL is an Android-specific language used to define the methods exposed in the `Stub` that can be reached from the `Proxy`.

B. Known Potential Pitfalls

The complexity of system services opens to many potential vulnerabilities. One specific aspect that has been explored by previous works relates to inconsistencies in the placement of security checks like permission enforcing or identity control [37], [25], [14]. The common root cause is that the checks were performed only in the Manager and not also in the Service counterpart. Thus, a malicious app could use a lower-level Proxy to communicate directly with the Service, bypassing the security checks. All these existing vulnerabilities have been fixed by Google and do not pose a threat in recent versions of Android. However, we show that this "layered" architecture still leads to new challenges and that they play a key role when looking for APIs vulnerable to state inference attacks. While the layered architecture is known to create problems in terms of placement of security checks, we believe we are the first ones to show how this complex architecture affects other security aspects as well.

V. Technical Challenges

One key contribution of this paper consists in the design and implementation of an automatic framework to identify vulnerable APIs leading to state inference attacks. This section discusses an overview of the several technical challenges we faced while designing this system, most of which have been overlooked by previous works and were a direct cause of false negatives (see Section VI-F and VII-E for a direct comparison with SCAnDroid [27]).

Enumerating the attack surface. The first key challenge is to determine the effective attack surface available to a potential attacker. Past works analyzed client- and server-side APIs and they highlighted security-relevant differences [37], [25], [14]. However, we show that there are server-side APIs (available to an attacker) that *do not have their associated client-side API.* There is thus a "hidden" layer of APIs that has not been considered by previous works. This makes previous approaches that enumerate the attack surface by only checking the client-side API significantly incomplete. In fact, in an attempt to quantify how much attack surface is "missed" we performed static code analysis on the Android framework itself and found that, *in the best case*, only about 44% of the attack surface is considered (see Section VII-E for the details).

Argument creation and validation. When directly invoking server-side APIs, one has to determine how to create "valid" arguments, otherwise the API may just return an error. We also note that, by interacting with the server-side API, one has even more flexibility in terms of argument creation since the client-side-only sanitization routines (if any) are bypassed. However, creating a successful object automatically is not so immediate and hides many challenges. For example, even a single field of a complex object, if not initialized correctly, can lead to the generation of exceptions with the risk of completely blocking the automatic analysis process.

System stimulation. Another important challenge consists in properly stimulating the system to induce the information leak. It is important to give, or create, the chance to the vulnerable APIs to actually leak sensitive data.

Systematic inspection of return values. One last overlooked challenge relates to how properly inspect values returned by an API. Previous works have relied on invoking every public (and private) method of the returned object, hoping to access fields that could be interesting for an attacker. However, this approach has several problems. First, the proper order of the invocations is unknown and may make a difference: for example, invoking a setter method before a getter method may cause the field value to be overwritten and permanently lost. Second, a client-side API may have access to some security-sensitive information, but it may "sanitize" the information before returning it to the caller. Even if the sanitization is not present, there can be private fields that are not accessible via the object's methods — not even the private ones. We found that, if not handled properly, this is yet another direct cause for false negatives.

VI. Analysis Framework

This section introduces our new analysis framework. We start by presenting an overview, we then discuss the various analysis steps and how we addressed various challenges, and we then offer a direct comparison with the most recent related work, SCAnDroid [27].

A. Overview

Our analysis framework is constituted of several steps, each of which tries to solve one of the challenges listed above. The first step enumerates the attack surface and its APIs (see Section VI-B). Then, we analyze each API to determine if it leaks sensitive information about other apps. The framework starts invoking it several times while keeping the system "at rest" (i.e., without performing any

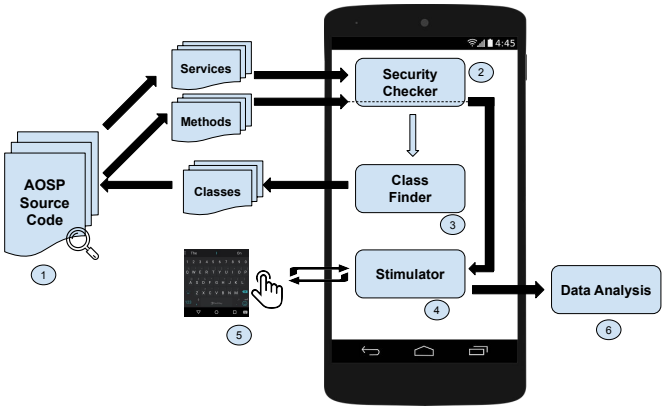


Fig. 3: This figure represents the framework and how the different components interact with each other.

other operations). Then, it starts a victim app (the actual app used for this part of the experiment is not relevant), while it keeps repeatedly invoking the API under analysis — and logging every invocation and every returned object. To conclude, we post-process these logs to identify potential correlations between the returned value of a given API and changes in the surrounding environment (e.g., the moment in which the victim app has been started). The output of the analysis system is the list of APIs that could be potentially used to mount state inference attacks.

Analysis framework organization. The framework is composed of six different modules. Figure 3 provides a detailed overview. First, it *enumerates the attack surface*: this process involves three modules: the `Extractor` (①), which extracts Android services; the `SecurityChecker` (②), which removes "candidates" (i.e., Services or Methods) that are causing any kind of Security Violation when invoked; and the `ClassFinder` (③), which, for each service, extracts from the device the name of the classes implementing it (both Managers and Interface). Once the services and classes have been enumerated, the `Extractor` and `SecurityChecker` modules extract and analyze all the methods implemented by these classes. All methods whose invocations do not cause `SecurityException` are then automatically invoked by the `Stimulator` module (④), while, in the background, the `UI-Interaction Automator` module (⑤) injects different types of UI events to simulate a user starting a potentially sensitive app and her interaction. Finally, the collected results are processed by the `Data Analysis` module (⑥). Note that this last analysis step is performed off-line, while all the previous modules run on the device itself.

B. Enumerating the Attack Surface

The enumeration of the attack surface is perhaps the fundamental component of our system. Its correct identification, however, is not as easy as one might think and many challenges lie behind this complex process. To begin with, Android offers multiple ways to register and expose a service to applications. Moreover, there is not a single central location to locate all the services inside the source code tree. However, we note that all Android services should be exposed to the system by using one of the following methods: `addService`, `publishBinderService`, or `registerService`.

The parsing process is handled by the `Extractor` module and it is built on top of *JavaParser* [30]. We parse the source code and extract all the services that are statically included in the system by looking at the methods listed above. To avoid missing any reference to a service, we extract the services running on our test device using the "`service`" command-line utility. The two lists are then merged together. Other works used the same approach to enumerate and list the services available in the Android OS [25], [37], [14].

Note that a non-system app cannot interact with all services. Our threat model assumes the attacker has control over a non-system app that can request any non-system permissions: however, some privileged services are protected by strict SELinux policies or by some permissions that only system apps can request and any attempts to access them cause a `SecurityException` to be thrown.

To enumerate the services that are accessible by an attacker, we perform a dynamic analysis step: first, we grant all non-system permissions to our test app, then communicate with a given service while in the background we monitor for security exceptions and violations like SELinux runtime violation or security exceptions raised by missing permissions. For those services that we can interact with, we enumerate the methods accessible to an attacker. To this end, previous works [27] relied on the AOSP documentation. However, this documentation only exposes public client-side methods: this approach entirely miss the "hidden layer" of server-side methods that do not have their respective client-side one. In our work, we do consider client-side methods, but we extend this enumeration by considering server-side methods as well. Server-side methods are implemented starting from AIDL specifications. AIDL is an extension of Java and introduces some meaningful keywords that are adding information about the behavior of a given method. Since *JavaParser* is not handling AIDL as language, we wrote our own parser.

We note that, for certain aspects, AIDL is more expressive than Java. In fact, in AIDL, each method and arguments can be prefixed by so-called *keywords*. Among the many AIDL keywords, two of them are particularly important for our work. The first one is the `out` keyword, which specifies that an input argument "can be modified by the callee." This helps us recognize this argument as a potential output value. The second relevant keyword is `oneway`: it indicates that the method returns immediately after having sent the data, without "returning" a meaningful result. Thus, our analysis proceeds in discarding a method if 1) its return value is `void` and none of its arguments are marked with the keyword `out`, or 2) its signature shows that the method is defined with the keyword `oneway`. We note that previous works did not consider these possibilities, leading to yet another venue for false negatives.

As an additional filtering step, we also discard methods that have at least one argument of type `IBinder` since it is not possible, to the best of our knowledge, to obtain a reference to a valid `IBinder` token without relying on a permission granted only to system applications.

For each of the potential candidate methods, we repeated the dynamic analysis monitoring for security exceptions, and keeping for further analysis only the ones not throwing any security violation.

C. Stimulation Strategies

Once we collect the candidate methods, we then proceed to invoke them and analyze their returned values.

Semantics-aware arguments generation. Understanding the "semantics" of an argument can be very helpful to improve the effectiveness of this step. This analysis step considers information taken from source code type information and the argument names. Our analysis extracts the following arguments' semantics: 1) *App identifiers*: this category contains all the arguments identifying a specific app installed on the device, such as *uid* or *packageName*; 2) *Process identifiers*: arguments identifying a specific process running on the device, such as *pid*; 3) *Filesystem locations*: every storage volume in the filesystem can be identified by a specific UUID such as *storageUuid* or *volumeUuid*; 4) *Time values*: arguments related to time and time-ranges, such as *beginTime, startTime* and *endTime*.

Identifying (and properly supporting) these values allows us to maximize the likelihood that the target method will return something relevant since we passed an expected value. During our analysis, these arguments are initialized with specific values defined both statically or dynamically. For example, for arguments like *uid* or *packageName* we can statically define a value — such as the *uid* of the application we want to target. Instead, for arguments like *pid*, we need to retrieve them at runtime. Moreover, for time-related values, we enforce a "logical-constraint" such that *beginTime* will always be lower than *endTime*. For the rest of the arguments for which we do not have a semantic, we automatically instantiate objects with random content, as discussed next.

Generic argument creation. Even knowing the types of objects, it is not always trivial to create valid instances. In fact, the objects in the Android Framework can be very complex, and can contain many references to other objects, each of which must be correctly solved in order to correctly create the final object. To invoke a given method, all the objects necessary to perform its invocation must be properly created and instantiated: this is, of course, a process that, if done manually, would be time-consuming. We decide to adopt an automatic approach and instantiate all the objects using a *recursive algorithm* that tries to instantiate an object by iterating through all the available constructors and recursively tries to create a valid sequence of object to match at least one of them. We repeat the process for each nested object and for all objects belonging to the method's signature. All the *primitive types* and their corresponding *wrapper classes* are filled with random values.

Since a recursive approach may incur in *circular dependencies* and crashes due to the increasing size of the call-stack, we configure our system with a maximum threshold of five recursive calls. In cases of failure, we resorted to custom handlers. This was needed for 105 objects (2.4% of a total of 4,390). We note that this is a one-off effort (that does not need to be repeated for each version of Android). To the benefit of the community and future works, we will release all these custom handlers.

Argument generation strategies. The analysis runs in two configurations. In the first one, a method is invoked multiple times without changing the arguments. This means that, for every method, the arguments are created only once. In the second one, instead, arguments mutate at each invocation of the method. Having multiple configurations is important since this allows us to analyze different behavior. We identified situations where APIs were leaking sensitive information over time only with some particular arguments. For example, we found a vulnerable API leaking information only when one of its arguments (related to "time") was changing from the previous invocation to the new one. Testing this API without mutating the arguments would end up in wrongly marking it as "safe."

User-Interface interaction. Interesting APIs for an attacker are the ones leaking the current "state" of a target app. An app can be in *background*, in *foreground*, or transitioning from these states. In this work, we analyze the following subsequent states: when the app is not started yet, when the user opens it and interacts with it, when she stops it, and when she resumes it. The goal of this module is to inject multiple "events" while, in the background, the Stimulator repeatedly invokes the API currently under test. The automatic interaction with the UI tries to mimic the behavior of a real user. This module is built on top of `AndroidViewClient` [9], a library which helps the creation of "Android test automation" scripts [9].

D. Data Serialization

If the automatic creation of complex objects posed a challenge to solve, so it is creating a generic serialization method that can be applied to all objects returned the invoked APIs. To solve this problem, we implemented a custom serialization algorithm to store both *arguments* and *return value* collected by the Stimulator. The serialization algorithm follows the process we used to instantiate the different objects, but in a reverse order, following a *depth-first exploration strategy*. For each object, we start by defining a "child node" — represented by the actual object we want to serialize — we recursively dump all its fields and store both their name and value in a `key-value` format. For fields with "primitive" types, we store their textual representation, otherwise, we recursively apply the same algorithm to all its fields until we reach the "root" class, `java.lang.Object`. This allows us to unfold complex object in a flattened format — like `HashMap`. If we detect circularity, we only store the reference of the object without recursively analyzing it a second time — none of the known JSON serialization libraries support it. This technique allows us to have a very detailed representation of a given object, including all (possibly private) objects that it encapsulates, no matter its complexity.

E. Data Analysis

The last component of our framework is in charge of results analysis. Its main goal is to find which APIs can be used to mount state inference attacks. More in general, we want to automatically identify APIs whose return value is somehow influenced by the surrounding context and leak a meaningful value when the target application is going to be used by the victim.

Automatically identifying which APIs can potentially be used by an attacker hides many challenges. In fact, applying a too conservative approach may result in having a large number of false positives to analyze manually. The opposite problem is the case in which one adopts an overly restrictive approach, as there is the risk of eliminating a valid API and thus incurring in false negatives.

Our analysis, divided into two stages, represents a tradeoff. As part of the first stage, we start by considering all the collected API's return values. We consider the keys of these return values and we discard all keys whose value is constant across all the API invocations. It is safe to discard these keys because the attacker would not have any chance to infer any state-change just by observing a constant value.

Then, we identify those keys whose value is particularly noisy, i.e., the value has almost always a different value (e.g., out of 100 invocations, there are only a couple of repetitions). These values are likely not providing a strong signal for the attacker, but we opted to err on the safe side and we proceed to further inspection before discarding them. In particular, we empirically found that the

vast majority of these noisy values belong to one of the following categories: timestamps, incremental values (relative timestamps and auto-incremented sequential numbers), and pointers (i.e., memory addresses). We developed a simple, *entirely conservative* heuristics to identify whether a noisy value belongs to one of these categories, in which case, given their non-security-relevant semantics, we can safely discard them. We consider a key to be of a certain category if these conditions apply: *Timestamp* if, when all the values are converted in a "datetime" object, the dates are always compatible with when we run the experiments; *Incremental values* if, when we calculate the difference between each consecutive value, these differences are always a small positive number; *Pointers* if all the values, when interpreted as memory addresses, would point to memory locations within valid, mapped memory pages.

We stress that, if we cannot recognize the semantics of a noisy key, we do *not* discard it and we consider it as a potentially interesting.

At the end of this stage we obtain a set of candidate APIs for which at least one key has *not* been discarded; that is, these APIs have a chance to be useful for an attacker. We note that the APIs detected as part of this stage could already be interesting for an attacker, in the sense that these APIs are potentially returning a (changing) value that may be correlated with the outside environment. We then proceed to identify those APIs that can be used to determine state transitions of other apps.

The second stage is conceptually straightforward: we focus on identifying APIs that return the same value before the app has started, and that suddenly start returning a different value just after the user (in our case, the `Stimulator` module) has started the victim app. The resulting APIs are the final output of the analysis pipeline.

F. Comparison with SCAnDroid

As we mentioned throughout the paper, we are not the first ones to propose an analysis framework to pinpoint Android APIs vulnerable to state inference attacks. This section offers a direct comparison with a recent work with a similar goal, SCAnDroid [27], and we show how it overlooked several of the challenges discussed in Section V. Most of these shortcomings are not just implementation issues, but they are about important aspects that were not considered.

The first group of shortcomings relate to how SCAnDroid determines the set of potential APIs to test. First, they only consider client-side APIs — the ones implemented in the Manager, by relying on the AOSP documentation. Our analysis, instead, considers a wider attack surface — the full list of methods exposed by client-*and* server-side components. We determined the list of APIs to test by relying on the source code of the AOSP project. Second, to limit the number of APIs to analyze, SCAnDroid performs a filtering step by only considering APIs whose name starts with a prefix such as *get*, *query*, *has* or *is*, assuming that only similarly named methods could constitute vulnerabilities. Our filtering process instead is based on the internal functioning of the Android system.

Our evaluation shows that these strategies allow SCAnDroid to potentially reach only ∼44% of the available attack surface (see Section VII-E for the detailed comparison on the final results). Last, we note how SCAnDroid cannot be easily extended to identify and support the test of server-side APIs, the ones reachable only via AIDL. These APIs are not accessible neither via *reflection*, nor are described in the official documentation available to the developers. These two techniques are the ones used by SCAnDroid to enumerate the attack surface. Thus, it is conceptually and technically not possible for SCAnDroid to cover and analyze this important portion of APIs.

Description	Android 8.1.0	Android 9
Available Services	160	180
Proprietary Services	2	16
`SELinux` Denials	35	44
`Runtime Permission` error	2	1
Unreachable Services	9	14
Native Services	12	10
Attacker-Reachable Services	100	95

TABLE I: Extraction of attacker-reachable services.

One other conceptual limitation relates to the limited ability to invoke APIs with proper arguments (e.g., pass a valid *process id* when needed) and, more importantly, how it inspects the return values. In fact, SCAnDroid recursively invokes all methods implemented by the returned object through *Reflection*, leading to two conceptual problems. First, the *order* these methods are invoked with may permanently modify the return value and some data may be lost. For example, invoking a *setter* method before the *getter* method of a specific field overwrites the field's value, potentially losing information. Second, and more importantly, *there is no guarantee that all information stored in an object are accessible via its public or private methods*. Our approach, instead, relies on a custom serialization that can recursively dump *every* field that is directly or indirectly stored within a given object, thus solving the problems of the previous approach. Our analysis found that these conceptual limitations are the direct cause of false negatives for SCAnDroid. In fact, our approach identified vulnerable APIs that were either not analyzed or for which the analysis wrongly marked them as "not vulnerable."

VII. Evaluation

A. Experimental setup

We evaluate our framework's efficacy on two versions of the Android OS: Android 8.1, running on a Nexus 5X with the latest available security patch (with security patch, December 2018), Android 9, running on a Xiaomi MI A2 (August 2019). Finally, we also tested our system on the latest version available at the time of writing, Android 10. However, we noticed that our system was not able to identify any new vulnerability on this latest version, despite the fact that the attack surface had been correctly identified and several APIs had been tested. Moreover, we have also manually verified and confirmed that all bugs we identified affecting Android 8.1 and 9 have been correctly fixed on Android 10. Thus, since no additional vulnerabilities were found on Android 10, in the rest of the section we will focus the discussion and the analysis of the results obtained on Android 8.1 and 9 versions.

B. Attack Surface Enumeration

Attacker-reachable services. Our system extracted a total of 160 services for Android 8.1. After having applied the filtering steps

# Methods	Client-side		Server-side	
	v. 8.1.0	v. 9	v. 8.1.0	v. 9
Total	3,536	4,092	2,683	2,887
After static analysis	1,080	1,324	1,384	1,472

TABLE II: The table summarizes the filtering process applied on the APIs extracted from the AOSP source code, as described in Section VI-B.

# APIs	Fixed Args		Mutated Args	
	v. 8.1.0	v. 9	v. 8.1.0	v. 9
Total	2,464	2,796	2,464	2,796
Accessible by an attacker	1,616	1,931	1,614	1,929
By removing constant APIs	813	1,127	816	1,141
By removing noisy APIs	48	35	51	52
Unique Methods	66			
Potentially vulnerable	24			

TABLE III: The table summarizes the filtering process based on the two stages described in Section VI-E.

described in Section VI-B, it identified how a non-system app can interact and reach 100 of them ($\sim 62\%$).

For what concerns Android 9 instead, we identified 180 services, but only 95 reachable ($\sim 52\%$) from an unprivileged app.

As it is possible to see, for both versions of Android, the majority of the services not reachable by a third-party application is due to security violation. By monitoring these denials, in fact, our system identified how more than the 23% of the services for Android 8.1, and 25% for Android 9.0, were not reachable by a third-party application due to missing permissions or SELinux violations. This first filtering procedure applied to services has allowed our system to extract only those services that can actually be used by an attacker.

Table I shows how many services were not reachable and for what reason.

API enumeration. Starting from the extracted services, we then proceed by identifying and extract first the Manager and the server-side services implementation, and then the candidate APIs.

On Android 8.1, the 100 services define a total of 157 classes. These classes are divided in 71 Client classes ($\sim 45\%$) and 86 Server ($\sim 55\%$). From these 157 classes, we identified a total of 6,219 invocable methods. These are all the methods that can be potentially used by an attacker to mount state inference attacks. We then proceed by applying the filtering rules, as described in Section VI-B. This process allowed us to obtain, from the initial bucket of 6,219 methods, 2,464 candidates to test on Android 8.1. Out of these 2,464 methods, 1,080 are exposed through the Client while the remaining 1,384 are available from the Server. We then dynamically tested all these methods looking for security violations. These methods have to be discarded since a third-party application cannot invoke them. This stage identified how only 1,616 of them is effectively reachable by a third-party application. Thus, the combination of both static and dynamic analysis reduced the candidates from 6,219 to 1,616.

We then applied the same identification and filtering process to Android 9. For what concerns this version, from the 95 initial services, we extracted a total of 157 classes: 76 acting as Client ($\sim 48\%$) and the remaining 81 as Servers ($\sim 52\%$). From these classes, we then extracted a total of 6,979 invocable methods. The first static filtering allowed our system to extract, from the 6,979 methods, 2,796 candidates (1,324 methods declared in the Client, while 1,472 defined in the Server). Instead, by removing the methods raising a security violation at runtime when invoked, our system pinpointed 1,931 methods effectively reachable by a potential malicious application. Thus, the combination of these pruning strategies allowed us to lower the number of methods to test from 6,979 to 2,796.

Table II and Table III summarize the results obtained during these pruning stages.

C. Method Testing

We analyzed each method for an average of 70 seconds (60 seconds plus time used for booting with both the configurations of the Stimulator). The overall execution time to run all the experiments is of *63 hours* for Android 8.1, while it took *68 hours* for Android 9.

D. Analysis Results

We then proceed to analyze the data collected during the tests. We start by discarding APIs not leaking any sensitive information due to their values remaining constant, as well as very noisy APIs, as described in Section VI-E. This process drastically reduces the number of APIs to analyze in the second stage. For Android 8.1, we reduced the number of APIs from 1,616 to 51 — *discarding ∼96.6%*: for Android 9, we started from 1,931 APIs and we ended up with 52 candidates — *discarding ∼97.5% of APIs*. In total, we obtained 66 unique APIs whose return value change appears to be conditioned by the surrounding context. Out of the 66 APIs, the second stage of the algorithm identified 24 potentially leaking APIs that can be used to determine whether a target app went to "foreground." Table III summarizes all the intermediate results of these filtering stages.

Out of these 24 APIs, 18 are indeed vulnerable: 4 APIs require no permission at all, *2 require a permission marked as Normal*, while the remaining APIs are protected with the `PACKAGE_USAGE_STATS` permission, which allows an app to collect the usage statistics of other apps, *including the application in foreground*. This information, as discussed in Section II-B, is of essence when mounting phishing attacks. Table IV, reports more detailed information about these APIs.

We now discuss the 6 false positives. Interestingly, two APIs actually leak *some* information about the surrounding system: `getInputMethodWindowVisibleHeight`, which returns the size of the keyboard on the screen, and `getPendingAppTransition`, which tells the attacker that an application "is going to be moved on foreground." The attacker can reliably infer that *an* app is about to change its state, but she cannot determine *which* app. However, since this scenario could lead to a more generic phishing attack, we conservatively consider these as false positives. For example, with the `getPendingAppTransition` API the attacker can evince that the user is about to interact with an app: Thus, she can simply display a pop-up a message informing the user that an update is available (without the need of specifying the name of the app). Since the timing is perfect, the user might be lured into clicking it. The same attack can be mounted with the `getInputMethodWindowVisibleHeight` API. In fact,

Classname	Method	Permission	Hidden	Affected versions	Fixed?
IActivityManager	isAppForeground	None	**Yes**	Both	**CVE-2019-9292**
IActivityManager	getProcessPss	None	**Yes**	Both	**CVE-2020-0087**
ActivityManager	getProcessMemoryInfo	None	No	Both	**CVE-2020-0372**
IUsageStatsManager	isAppInactive	None	No	Both	**CVE-2020-0317**
INetworkStatsService	getUidStats	ACCESS_NETWORK_STATS	**Yes**	Only 9	**CVE-2020-0327**
INetworkStatsService	getDataLayerSnapshotForUid	ACCESS_NETWORK_STATS	**Yes**	Both	**CVE-2020-0343**
StorageStatsManager	getFreeBytes	None	No	Both	**Duplicate**
StorageManager	getAllocatableBytes	None	No	Both	**Duplicate**
IActivityManager	isUidActive	PACKAGE_USAGE_STATS	**Yes**	Only 9	Won't fix
NetworkStatsManager	querySummary	PACKAGE_USAGE_STATS	No	Both	Won't fix
NetworkStatsManager	queryDetailsForUidTagState	PACKAGE_USAGE_STATS	No	Only 9	Won't fix
IActivityManager	getUidProcessState	PACKAGE_USAGE_STATS	**Yes**	Both	Won't fix
IActivityManager	getPackageProcessState	PACKAGE_USAGE_STATS	**Yes**	Both	Won't fix
NetworkStatsManager	queryDetailsForUidTag	PACKAGE_USAGE_STATS	No	Both	Won't fix
UsageStatsManager	queryEvents	PACKAGE_USAGE_STATS	No	Both	Won't fix
UsageStatsManager	queryUsageStats	PACKAGE_USAGE_STATS	No	Both	Won't fix
UsageStatsManager	queryAndAggregateUsageStats	PACKAGE_USAGE_STATS	No	Both	Won't fix
IStorageStatsManager	queryStatsForUid	PACKAGE_USAGE_STATS	No	Both	Won't fix
IStorageStatsManager	queryStatsForPackage	PACKAGE_USAGE_STATS	No	Both	Won't fix
NetworkStatsManager	queryDetailsForUid	PACKAGE_USAGE_STATS	No	Both	Won't fix

TABLE IV: Systematization of the vulnerable APIs. For each API, we report the vulnerable service, the type of permission protecting it, if the API was present in the Manager or only in the Proxy component, which version contains the vulnerable API and if the bug has been fixed.

the attacker can infer when the user is going to use the keyboard, giving her the chance to show a popup informing the user that a keyboard update is available.

Two other APIs (*createAppSpecificSmsToken* and *DownloadManager.Query*) return very noisy values, which change at every invocation. We note how the filtering step described in Section VI-E does not discard these APIs because the noisy values are not belonging to one of the categories known to *not* leak information (e.g., timestamps). In fact, a deeper analysis of these two APIs allowed us to confirm that their return value is either a *pseudo-random token* (for the first API) or an *object identifiers* (for the latter). Neither of the API, thus, return a value correlated with the current state of the system.

The last two APIs are *launchLegacyAssist* and *getAllCellInfo*. Their values changed after the start of the target application but it does not appear to be correlated to the target app's state transition.

For completeness, we manually inspected the remaining 42 APIs out of the 66 that have been filtered out by the second stage. We identified how 7 APIs leak "system state" information, such as the total amount of bytes written by apps, or aggregate statistics about the disk usage. 15 APIs, instead, leak sensitive network information, like the overall network usage. We found that the remaining APIs do not seem to leak any relevant information.

An interesting observation comes from the vulnerable 3 APIs affecting only Android 9. In fact, they are all new features introduced in existing services, which were also available in Android 8.1. *This continuous evolution underlines the importance of having an automatic analysis tool to flag these potential problems.*

Disclosure. We disclosed our findings to the Android security team. *Six* APIs have been acknowledged and fixed by Google and a CVE was assigned. Table IV provides a detailed list of the APIs fixed and the assigned CVE. We believe this confirms how seriously Google is considering this class of vulnerabilities. For what concerns the remaining APIs, the Android security team considered them as "won't fix" due to the type of permission protecting the API. However, it is important to highlight how these APIs are exposing to the attacker sensitive information about the state of the apps running on the phone. Moreover, we note how real-world malware already abuse similar APIs that require the same permission, as documented by recent findings by security companies [15], [20]. We believe it would be possible to secure these APIs by adjusting the granularity of the information returned.

E. Results Comparison with SCAnDroid

To further illustrate the performance of our system, and to show how our contributions play a key role on the automatic identification of state inference vulnerabilities, we compare our results against those obtained by SCAnDroid on the same Android version — Android 8.1.

Overall, our system was able to correctly detect *all* the vulnerable APIs identified by SCAnDroid. However, we note that most of the vulnerable APIs identified by SCAnDroid belong to bugs that, in this paper, we categorized as leaking information related to the *system* (like the total amount of bytes written by apps, or aggregate statistics about the disk usage) and *network* states (like the overall network usage), as described in Section VII-D. While these bugs are interesting and they can be exploited with *template attacks*, as showed in [27], it is not trivial to weaponize them.

Our approach focuses on finding vulnerable APIs—and this is one first difference with SCAnDroid—such as those ones that allow

an attacker to pinpoint which app the user is currently interacting with, or that at least do not require building "templates" for each target victim app. As presented in Table IV, only two of the bugs we found were marked as *Duplicate*, while all the other ones were previously unknown. All the APIs identified by our system are generic and are not related to a specific feature or configuration of a specific application, making our findings more generic and scalable.

Extending the attack surface allows us to examine components and methods that were not even taken into account by SCAnDroid. To determine how many methods SCAnDroid missed, we identified the server-side methods that are *not* reachable from the Managers. To collect this number, we first extracted all the server-side methods defined in the Android OS, version 8.1, obtaining 5216 methods. Then, we extracted "interesting candidates," as described in Section VI-B: we identified that only *1,384* of them are actually potentially reachable by an attacker and thus represent the attack surface analyzed by SCAnDroid. Since SCAnDroid uses as entrypoints only a subset of "client-side" methods, we then determined how many of the 1,384 methods are effectively reachable from the Managers. To this end, we computed a forward callgraph for each of the methods defined in the Managers. If one considers only client-side methods, only *835* methods, out of the 1,384, are potentially reachable ($\sim$60% of the attack surface). However, SCAnDroid does *not* take into account *all* client-side methods, but it applies a filtering process based on the method's name. We applied the same filtering process on the client-side methods and found that SCAnDroid would be able to reach only *616* server-side methods, which is *only about the 44% of the attack surface*. We also note how, for what concerns Android 8.1, the 33% of the bugs we identified (5 out of 15) resides in the server-side component. This shows, once again, that the server-side attack surface should not be overlooked.

To conclude, our analysis correctly detected 10 vulnerable APIs that satisfied SCAnDroid's filtering. Thus, these APIs have been tested, but were not marked as vulnerable. All 10 APIs are present in Android 8.1, are exposed in a Manager and match the prefixes constraints that would pass SCAnDroid's filter (e.g., *getProcessMemoryInfo*, *queryUsageStats*, or *queryAndAggregateUsageStats*). We believe that a possible explanation relates to how SCAnDroid stimulates the APIs or how it processes the return value. An emblematic case is *getProcessMemoryInfo(int[] pid)*. This API leaks statistics about the memory usage of running applications. However, to detect this leak, the API needs to be invoked with a list of valid "process id," otherwise a set of `NULL` is returned. We believe SCAnDroid might have misclassified this API due to not passing proper arguments. Since our system identified "pid" as a *meaningful argument*, our analysis handles this case and spots the vulnerable API. This is another important result that shows how the argument generation we applied, described in Section VI, improves the effectiveness of the identification of vulnerable APIs.

VIII. Case Studies

This section discusses three case studies to demonstrate how the vulnerable APIs we identified can be used to mount phishing attacks. We opted to discuss specific instances of vulnerabilities highlighting three different categories of problematic APIs. For the interested reader, Appendix XII-A reports concrete proof-of-concepts on how these APIs can be exploited in a real attack scenario. Note that to prove the feasibility of exploitation of all the vulnerable APIs identified by our system and listed in Table IV, we provided to Google, during the disclosure process, a Proof-Of-Concept for each API to show how it can be used to infer which application is going to be used by the victim.

isAppForeground - CVE-2019-9292. This API is implemented by the `ActivityManager` system service: it takes as argument a Linux user id (UID) and it returns a boolean indicating if the app run, by this user, is in foreground. Since in Android each installed app is assigned a different UID, and since the UID $\rightarrow$ mapping can be easily obtained, an attacker can invoke multiple times the API to check when the target app goes to foreground (the proper time to spoof its UI). This API thus represents the "ideal" case for an attacker, as she can monitor the state of any app installed on the device. This API does *not* require any permission.

getDataLayerSnapshotForUid - CVE-2020-0343. This API is implemented by the *NetworkStats* system service, and it is only available through the AIDL interface. This API takes the *UID* of a target app and it returns a *NetworkStats* object encapsulating network statistics for said app. Our framework identified multiple fields leaking sensitive information; two of them — namely *set* and *txPackets* — can be used in combination to successfully mount a state inference attack. The *txPackets* field indicates how many packets the app transmitted since the boot, while the *set* field indicates whether the packets are sent while in foreground. When the malware notices an increment of *txPackets*, in conjunction with a change in the *set* field, it can infer that the target application is performing, for example, a login, and can react accordingly. This API requires the `ACCESS_NETWORK_STATE` permission: since this permission is "normal," it is silently granted at installation time.

queryEvents. This API is implemented as part of the *UsageStats* system service. It takes as input a range of time and returns a *UsageEvents* object, which embeds information about all the events triggered by the apps running during that time span. Our framework identified a number fields leaking information about the state of an app, which, if combined together, can be used to mount a state inference attack. In particular, an attacker can combine *mPackage*, that indicates the package name of the app performing the "event," and *mEventType*, that specifies the type of the event. Note that other combinations are effective as well. In this case the attacker is interested in monitoring for a `MOVE_TO_FOREGROUND` event, which indicates that the app moved to foreground, the ideal moment to show the spoofed UI. This API requires the `PACKAGE_USAGE_STATS` permission, which the user needs to manually approve. Nonetheless, *real-world malware has been found in the wild that had the same exact requirements, showing that this request is legitimate* [20], [28], [15].

IX. Detecting State Inference Attacks

We believe that automatically identifying APIs that make the system vulnerable to state inference attacks is a good first step forward to eradicate this problem. However, all existing techniques combine static and dynamic analysis, which potentially open these approaches to false negatives. To protect users from unknown vulnerabilities, we studied the feasibility of an additional component, which aims to be *a runtime defense and detection system to identify state inference attacks at the moment they occur*. The design of this component is based on the following two intuitions.

The first one, which is somehow well known, is that *all* existing state inference attacks need to implement *polling behaviors*. With this term, we refer to an application invoking multiple times a set

of APIs within a short time window. Malware exploiting vulnerable APIs to mount state inference attacks *need* to use polling to ensure they can *race* the target app and make their spoofed UI appear on top at the right time.

The second intuition, which, to the best of our knowledge, has not been explored before, is based on the following key hypothesis: *benign apps rarely rely on polling and, when they do, the nature of their behaviors is different than those of malicious apps.* Our hypothesis, if verified, would consequently imply that the polling behavior could be used as a *strong indicator* to distinguish between malicious and benign apps, where with "strong indicator" we refer to a signal that would not lead to an unacceptable amount of false positives.

This section is organized as follows: 1) We present and discuss the results of the analysis on a dataset of malicious apps. The aim of this analysis is to identify peculiarities in terms of APIs invocation frequencies adopted by phishing apps (§IX-A); 2) we perform an analysis on a dataset of about 2K benign apps: this acts as our "training set" to verify the hypothesis mentioned above (§IX-B); 3) We used the collected insights to guide the design of an on-device detection system (§IX-C); 4) We discuss the implementation of the system, and an evaluation on a different dataset of 8K benign apps (which acts as our "testing set"), and its performance (§IX-D); 5) we compare our work with the most closely related work, LeaveMeAlone [38] (§IX-E).

A. Peculiarity of Phishing Applications

To verify the validity of our hypothesis, we first perform an empirical study on malicious applications. For this study, we selected a dataset of 50 samples from *all* the families of Android malware that 1) were discovered in the last four years and 2) are known to mount state inference attacks. In particular, we analyzed samples and variants from: `Anubis`, `LokiBot`, `ExoBot`, `BankBot`, `RedAlert`, `MisteryBot`, `BianLian`, `Asacub`, and `Gustuff` [15], [28], [32], [21], [17], [20], [29], [16], [31]. For each family, we analyzed both "malware-only apps" — apps containing only the malicious code — as well as "repackaged apps." Analyzing sophisticated malware is not always an easy task: we encountered different situations that made the (automatic) dynamic analysis very challenging. In these specific cases, for example, we found apps performing integrity checks on the device or anti-hooking techniques, as well as starting the malicious behavior only after some time or after certain actions, probably to avoid Google Bouncer analysis. Moreover, many samples tried to communicate first with a remote C&C server: since most of these servers were "unreachable" at the time of test, the malware did not start any activity. To overcome these difficulties, we decided to manually analyze the samples looking for the code in charge of performing the state inference attack. For each family, we extracted the methods used to perform this task. Our analysis highlighted different techniques used to mount this attack. To perform polling, malware authors are using different mechanisms like registering a repeated-delayed task with `postDelayed()` or `AlarmManager`. Another technique relies on anonymous `Thread` or `IntentService` to invoke the vulnerable API *every second*. Lastly, an even more aggressive technique consists in executing all the "monitoring logic" inside a `while loop`, without any delay between invocations. It is possible to model and define a common behavior shared among all the families we analyzed: we found that *all malware poll with a maximum delay that spans from 600ms to one second* (i.e., a frequency of at least 1Hz) and that a *malware never stops this behavior once it is started* (i.e., polling is performed for a "sustained" amount of time).

During the years, malware evolved and changed frequently the set of vulnerable APIs and techniques used to identify the starting of a sensitive application to target with a phishing attack. The techniques used by a malware highly depend on the API level the device of the victim is targeting. For example, if the device targets an Android lower than 5.0, the malware will adopt a combination of both `getRunningTasks(int)` and `getRunningAppProcesses()`. Instead, if the device targets a version between 5.0 and 6.0, then the malware can still rely on the information exposed by the `proc` filesystem (`/proc`).

However, as discussed in Section II-C, Google fixed all the known components leading to a leak of sensitive information like the state of an application. Hence, the only available attack vector for the malware is to rely on the APIs protected by the well known `BIND_ACCESSIBILITY_SERVICE` permission (`a11y`) [12]. As it is possible to see, some sophisticated malware like `Bankosy`, `Cepsohord`, and `MysteryBot` started moving from the `a11y` towards exploiting vulnerable APIs protected by the `PACKAGE_USAGE_STATS` [20], [35], [5]. This transition might also be forced by the fact that Google is going to remove all the applications using the `BIND_ACCESSIBILITY_SERVICE` permission for anything except helping disabled users [8].

Moreover, [35] highlighted how the adoption of the `PACKAGE_USAGE_STATS` permission amongst malicious applications published on the official Google PlayStore is growing. This is an important result showing that, even if Google is not going to fix the vulnerable APIs we identified in Section VII, they are used by malware developers in real-world attacks [20], [35].

The `PACKAGE_USAGE_STATS` permission, like `BIND_ACCESSIBILITY_SERVICE`, can only be granted through the `Settings` application: this means that a malware cannot ask at runtime this permission. However, as for the attacks based on `a11y`, the malware can directly display the Settings application and lure the user through social engineering to grant the permission. As presented in [5], malware uses social engineering while masquerading as Google Chrome by mimicking the application's icon and name. This technique tricks the victim into thinking she is granting the `PACKAGE_USAGE_STATS` permission to the Google Chrome app, while instead, she is granting the permission to the malicious app.

B. Peculiarity of Benign Applications

As the next step, we characterize whether and how benign apps perform polling-like behavior, and whether there are some features that can be used to distinguish them from malicious attempts. To this end, we built a dataset of 10,108 benign apps. To select a representative dataset, we consulted AndroidRank [3] to find popular apps, which we then crawled from the Play Store. The resulting dataset is constituted as follows: 9,066 "top apps" with at least 50M installations, while the remaining 1,042 were chosen randomly from apps with a number of installations ranging from 10M to 50M. From this dataset, we built two different datasets: a "training set" of 2,042 apps (roughly 20% of the dataset), and a "testing set" with the remaining 80% of it.

The rationale behind this choice is the following: we first investigate how benign apps perform polling by *only* considering apps within the *training set*. Based on the insights of this step, we then 1)

enumerate a number of observations that can be used to distinguish between benign and malicious samples and we use them to build a detection system; 2) we evaluate the performance of the proposed system (in terms of miss detections) by analyzing the apps in the *testing set* — which are *not* considered during the design/training phase. We believe this two-step approach helps addressing concerns related to how our evaluation would generalize to a bigger dataset.

Testing Environment. To study the runtime behavior of benign apps, we instrumented the Android OS (Android 9 running on a Pixel 3A) to log all binder communications and filesystem activities for a given application. This log contains information such as the service and the API invoked by the app, and the correspondent timestamp.

Analysis System. To identify a "polling-like behavior," we tuned the analysis to flag all the syscalls and APIs invoked at a rate of *at least once every two seconds (i.e., 0.5Hz), for at least 60 seconds*. We believe these thresholds are a "safe assumption," since 1) the threshold frequency is twice as low as the minimum frequency rate at which malware performs polling activities (i.e., 1Hz) and the phishing attack would necessarily incur a delay of 2 seconds, making it visible to the victim; and 2) the malware does not stop polling activities after it has started it (see Section IX-A).

A very important aspect of the proposed system is that *it does not look for polling by just considering a single API, but it considers the overall number of invocations*. That is, instead of monitoring whether a specific API A is invoked more frequently than once every two seconds, the system monitors if the app has cumulatively invoked *any* API more frequently than our threshold. This design choice introduces the concern of false positives (which we fully address in the remaining of this section), but prevents an attacker to bypass our detection by simply alternating the invocation of two (or more) different APIs, thus lowering the per-API frequency.

We now report the results of this analysis. We also note that this analysis system, configured with the thresholds we mentioned, is able to detect all the malware samples in our dataset.

Results and Observations. We now discuss the results and the observations after the execution of each of the 2,042 apps of the training set within our instrumented environment. We executed each app for five minutes. We post-processed the execution traces on our analysis system to identify if also benign apps perform polling, and, if so, on which component and at which frequency rate. From the results of the analysis, we draw the following two observations:

1) Benign apps do perform polling. We found a significant number of apps that were flagged by our system. More interestingly, we analyzed the traces to identify which APIs were being flagged and we identify frequent patterns belonging to the following categories: *a) Graphical User Interface*: to draw the content of the app's view, the system relies on polling to design the various component forming the UI of the app; *b) Audio and Video*: similar to the GUI, multimedia components also rely on polling. In fact, to reproduce the audio and video stream, the multimedia services needs to refresh, for each frame, the video and audio buffer. *c) DRM*: when playing rights-protected content, the DRM service first decodes and then forwards to the multimedia service each chunk of the file to play. *d) System Services Internals*: operations that are performed each time a system service is used by an app. For example, when an application interacts with a system service that operates on global data, a new Thread is started and multiple `acquireWakeLock` and `release-WakeLock` APIs are invoked to handle tasks synchronization. In all these cases polling is performed by system services "on behalf of the app." That is, even though the polling logic is implemented in the system service, it is still related to the context of the app since the service uses the app's identity for the subsequent invocations.

We investigated each of these behaviors in detail, and we found that none of these APIs can lead to abuse or state inference attacks, and we thus believe that they can easily and safely whitelisted. Table V (in Appendix) provides a very detailed list of our insights. We also note that the four groups above capture polling behavior for all apps in our training dataset except for six of them: these six apps were found to be App Lockers, which we discuss in Section IX-D.

2) Bootstrap phase. Another interesting observation is that we have noticed how apps often show a spike of activity during their "bootstrap time." This, intuitively, makes sense: when the app is started, it needs to perform a number of one-off setup operations, e.g., querying system information, setting up in-memory data structures, requesting permissions. However, we also noted how the level of activity (measured as the frequency of API invocations) decreases as the application transitions from its "bootstrap" to its "at rest" phase. We note how this characteristic is profoundly different from state inference attack malware behavior: *once the polling behavior is started, it is never terminated*.

C. Proposed Detection System

Based on the results of the previous empirical study, we implemented a system for the detection of polling behaviors on top of Android 9, by modifying the *execTransact* method of the *Binder* class, which is invoked any time a system service receives a request. This design choice prevents malicious applications to circumvent our detection system, since our modifications affect only the (privileged) server side of the Binder subsystem. Our system is setup to raise alerts for apps performing API invocations at a rate of at least x invocations per y seconds (with $x = 1$ and $y = 2$, i.e., a threshold minimum frequency of 0.5Hz), for at least z seconds (with $z = 60$, as previously discussed). Our system is also setup to not consider API invocations during a "bootstrap phase" of a given app, where with "bootstrap phase" we indicate the first k seconds from the app's start up. For our system, we empirically selected $k = 90$, but, for the sake of completeness, the evaluation section discusses how the accuracy of the system changes when k varies (between 0 and 5*60 seconds), and we show that this threshold affects the results in a minimal way.

Implementation-wise, the system creates a circular buffer for each running `uid` in the system. The length of each circular buffer depends on the number of invocations allowed in a given timeslot (x). We start the monitoring phase after the bootstrap time k, and we do not consider APIs that have been whitelisted (i.e., the "benign" and not-possible-to-abuse APIs discussed above belonging to one of the four categories). For each service invocation, our system stores the current timestamp in the circular buffer associated to the appropriate `uid`. When the circular buffer is full, the system checks whether the elapsed time from the first invocation in the buffer is lower than y seconds. Due to the properties of circular buffers, this is the case *if and only if* we have recorded x services invocations in less than y seconds. This means that the caller app has exceeded the invocations rate that we are interested in detecting. If the threshold is exceeded, our system enters a so-called "alert mode" and stores the time at which the polling behavior started in an additional variable (one for each `uid`). When handling the following invocations of the service while in alert mode, our system checks whether the polling behavior

is sustained for at least z seconds, and it does so by comparing the content of this additional variable with the current time. If the difference is greater than z, our system raises an exception (preventing the service's request to be completed) and it raises a warning to the user.

Note that if subsequent invocations do not meet the minimum threshold for polling, the system leaves the "alert mode" and it resets the internal state. We note how this system allows for the detection malicious apps performing state inference attacks polling on a single API, but, more importantly, it would also detect situations for which the malware uses multiple (different) vulnerable APIs to infer the state of the target application. This is possible due to using a "single bucket" for all APIs invoked by the same app (identified by their Linux `uid`).

To err on the safe side and to avoid false negatives, for our defense mechanism we set a very conservative detection threshold to *half the frequency of all real-world malware samples*. This allows us to detect all current malware samples analyzed in Section IX-A, and even if these malware samples would cut their polling frequency *in half*, our system would still detect them. In principle, a malware that reduces even more its polling frequency might bypass our detection system. However, to mount a successful phishing attack, *timing is a fundamental component*. Thus, lowering down even more than half the polling frequency, would make the malware and the attack ineffective, since there would be a very visible delay between the launch of the legitimate app and the spoofed one. For example, a situation where the user clicks on the legitimate banking app icon, and she starts to interact with the application, and only then, say after two seconds, the malware displays its spoofed banking app UI asking again for credentials, would certainly raise some warnings to the victim and the attack would be noticed.

D. Evaluation

We evaluated our on-device detection system on the testing set, composed by 8,066 apps. We stress that we did *not* access and/or inspect these apps before having finished developing the entire system. In other words, we believe this represents a realistic and fair evaluation on how our system would fair in practice. Our results show that the system would flag only 30 apps as potentially problematic, which represents only the *0.37%* of the entire dataset. We note that this result was obtained by setting a threshold for $k = 90$ to identify the bootstrap phase. To evaluate the impact of this threshold over the results, we varied it from zero seconds (i.e., we start monitoring the application as soon as it starts) to 240 seconds (i.e., we start monitoring the app 4 minutes after it starts): the number of false positives is not significantly affected — it varies from 39 to 25. Figure 4 shows a graph depicting how this number changes while varying the bootstrap phase threshold. We also note that this threshold does *not* affect the detection of malicious apps, since all malware samples never stop polling after they have started.

We now present a detailed analysis of the apps detected as problematic by our system. For this step, we consider out "worst case" — we consider the configuration that raised the highest number of false positives ($k=0$, 39 false positives). The goal is to analyze the polling behavior exposed by these apps and determine the nature of their behavior. We identified three groups of apps with similar patterns, which we discuss next.

The first group is composed by 10 apps polling only one of the vulnerable APIs we identified, `getProcessMemoryInfo`. In these apps, the code performing the polling belongs to a third-party

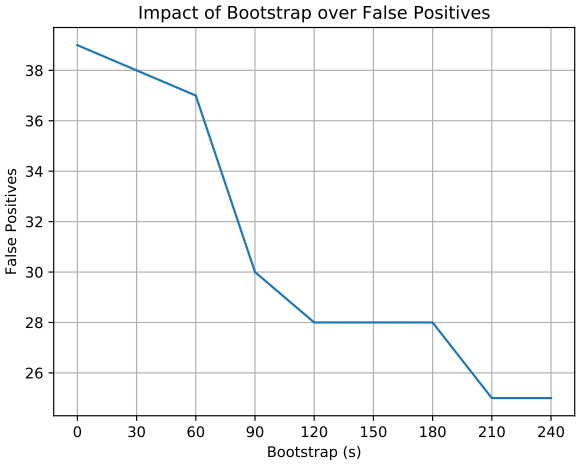


Fig. 4: The following plot shows the impact of the bootstrap phase in relation to the number of false positives identified by our system. As it is possible to see, when the bootstrap time is none, the number reaches 39 and it decreases as the application execution time increases. The lowest number of false positives is reached if the bootstrap phase is greater or equals to 210 seconds.

library for crash analytics that constantly traces the usage of the app's memory. However, this API is invoked to only monitors *its own* memory. We note that Google has now fixed this API, and it would allow an app to *only* monitor its own memory — making the usage of this API safely whitelistable.

The second group is composed of 10 apps, which embed ads libraries that aggressively poll several APIs to monitor the status of the network, probably to collect information related to nearby networks, with the goal of tracking the user [1]. We believe that users would be pleased to suppress this privacy-invasive functionality.

The third group of 9 benign apps is constituted by "App Lockers." These apps work by monitoring which app the user is interacting with, and by "locking" the device if the user is interacting with an app she should not interact with (e.g., the Settings app). These apps were initially popular as a way to protect the user phone, but they became less popular with time, and they are now considered "grey area." Google also introduced additional security features that make these apps of dubious utility. With that being said, these apps are problematic for our system as they do rely on polling (in this case, the `queryEvents` API), making this behavior indistinguishable from malware. Our system, as is, would block these apps — and rightfully so. If a user truly wants to use these apps, she can of course whitelist them. But given their declining popularity over time, we argue this is acceptable.

The last group is formed by 10 apps whose polling behavior is caused by bad coding practices: as a representative example, we identified an app continuously invoking the `getRunningServices` API with no sleep between two invocations.

For what concerns the malware detection, we evaluated our system over synthetic apps configured as real malicious applications. The techniques used to mimic the malicious behaviour of the apps are described in Section IX-A. For these apps, our system was able to correctly pinpoint the malicious behaviour of all the samples and

thus, in a real scenario, it would have been able to detect and stop the attacks.

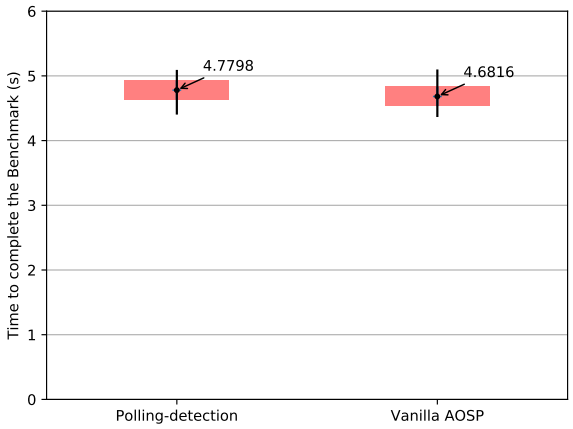


Fig. 5: Plot comparing the time needed to complete the benchmark for an unmodified AOSP system and one powered with our polling-detection system. The highlighted points are the arithmetic means computed over 100 runs of the benchmark. The red boxes represent their standard deviations, while the black lines indicate the minimum and the maximum times recorded for both systems. Our polling detection system is accountable for an overhead of ~98.2 ms per benchmark run (*1.98%*) in average, corresponding to ~9.82 μs per service API invocation.

Performance Consideration. The design of our detection system relies on optimized data structures and a fast algorithm. This allows our system to handle each service invocation in constant time (i.e., $\mathcal{O}(1)$), independently from the number of services in the system, the number of running apps, and the rate at which system services are invoked. An approximation of the required memory is given by $napps \times (x+1) \times 8bytes$, where $napps$ is the number of running apps and x the entries in the circular buffers. In an hypothetical scenario of 50 apps invoking multiple services, we estimate that our system needs in total less than 10KB of memory. We measured the performance overhead of our detection system over the vanilla version of AOSP by performing a micro benchmark, consisting in invoking multiple times the same system service and measuring the time needed for the system to handle all the requests. More in details, we invoked *ActivityManager* service's *getAppTasks* API for 10,000 times. We repeated the test 100 times for both our modified version and a vanilla version of AOSP. For the purposes of this benchmark, we modified our system to prevent it from raising exceptions when the polling threshold is surpassed: we do this to not invalidate the results of the benchmark, since returning an exception to the caller is much faster than actually invoking the API. Figure 5 (in Appendix) shows the results of the benchmark in terms of the average and the standard deviation of the time needed to serve 10,000 requests. In average, our detection system is responsible for an overhead of only the *1.98%* with respect to the AOSP baseline. We believe that such a low overhead is acceptable. Additionally, from a usability perspective, we did not notice any difference while using either a device running the baseline AOSP or our detection system.

E. Comparison with LeaveMeAlone

LeaveMeAlone [38] is a recent work whose main goal is to detect and block malicious applications performing a *runtime information gathering* attack on Android, and it is thus related to our work. This section discusses LeaveMeAlone in detail and it offers a direct comparison showing how it is affected by significant limitations when tasks to deal with phishing attacks.

A runtime information gathering attack consists in a malicious app stealing or inferring sensitive information about the runtime data computed in the context of a target application by analyzing the usage of shared resources. The core component of [38] is named "AppGuardian," which runs as an unprivileged application. It is in charge of monitoring the runtime behavior of the running apps and of detecting which are manifesting a suspicious behavior. To identify these apps, the system relies on collecting static information of the installed apps like suspicious permissions. For these suspicious apps, the system collects runtime behavior when they are running in background (e.g., thread names, CPU scheduling, kernel time). These behavioral information are collected by accessing the `procfs` subsystem. The identification of these suspicious apps plays a key role when a target app — protected by the Guardian — is started by the user: when this situation occurs, the system stops all the suspicious background processes by creating a "safe execution environment" for the target app: By not letting the suspicious apps running in background, the "runtime information gathering" attack is not feasible anymore.

Since our main focus is on detecting polling to prevent phishing attacks, one may think that LeaveMeAlone could be a good candidate to address the same problem. However, while LeaveMeAlone is certainly valuable in many situations, we argue it would be affected by many limitations when tasked to prevent phishing attacks.

First, AppGuardian *relies* on previously known vulnerabilities to collect runtime information regarding a specific app. Since it is designed to run as a non-privileged application, all (present and future) vulnerabilities of this kind will be eventually patched by Google [18], preventing this approach to work. As a case in point: *all* sources of side channels mentioned in the LeaveMeAlone paper have been fixed in recent versions of Android.

Second, we note that several of the vulnerable APIs found by our framework do not require any sensitive permission, making malicious apps using these bugs challenging to be detected by automatic vetting processes. Our approach, instead, only relies on the presence of polling-like behaviors and would detect these cases, independently from the requested permissions.

Third, AppGuardian heavily relies on whitelisting to make their approach work. Quoting the paper, "Overall, among all the popular apps, Guardian only needs to suspend 19.3% of the apps" when referring to a dataset of 475 apps, which is 92 apps. To avoid creating usability problems, the paper states that they rely on a whitelist: "The whitelist here includes a set of popular apps that pass a vetting process the server performs to detect malicious content or behaviors. In our implementation, we built the list using the top apps from Google Play, in all 27 categories." Our approach, instead, would only be affecting about 40 apps on a dataset of 10K dataset, which is 20x bigger than what used in previous work.

Last, AppGuardian is vulnerable to race conditions when tasked to detect on-going phishing attacks. In fact, both the malicious app and the Guardian are relying on the same side channels: if the malicious app wins the race (and detects a victim app has been

started), it can go to foreground *before* Guardian has a chance to kill it. However, once the malicious app is in foreground, Guardian does not have a chance to suspend it — third-party apps are not allowed to do so (they can only suspend apps that are in background). Our approach is not affected by this limitation. We reached out to the authors of [38], they acknowledged the presence of the race conditions, and they confirmed that, in this scenario, the Guardian is not able to stop the malicious app but only to inform the user with a notification.

We acknowledge that this comparison is a high-level one, but we argue that it is the best we could make, for multiple reasons. First, *all* the side-channels used are now fixed, and any evaluation would consequently show negative results. Second, a significant component of the LeaveMeAlone design is to rely on an off-market vetting system based on the detection of *dangerous permissions*. There are no details about this aspect in the paper and it would be very challenging to reproduce. Moreover, as shown in Table IV, we found several APIs that *do not* require any permission. Thus, once again, this would lead to obvious bypasses of the system. The last challenge that would limit any "more direct" comparison is due to the fact that the source code for neither the app nor the vetting system is available.

X. Limitations

We believe our work represents a step forward in the detection of vulnerable API leading to a state inference and to detect malicious applications exploiting these vulnerabilities to perform phishing attacks. However, we acknowledge that our approach is affected by the following limitations.

Reliance on availability of source code: Currently, our tool requires access to the source code of the Android framework. From the source code, it is possible to extract the semantics of the arguments, which is a fundamental step when creating argument values to invoke a given method. This, therefore, limits our tool to be used only in AOSP. Thus, our framework cannot be used to test systems from other vendors whose source code is not available, such as Samsung or Huawei. Note that, however, our system could be extended to bring the analysis at the *bytecode* level, and therefore would not require access to the source code. At the same time, we would lose important information such as the name of the arguments, which are used to generate meaningful values. To solve this last problem, our system could implement a more deterministic model in constructing and filling in arguments required for the polling APIs, making the system working on closed-source non-AOSP systems.

Challenges in detecting new phishing variants: At the moment, our on-device detection system can detect and stop the most classic of phishing attacks, the one in which the attacker infer which is the application that will be used by the victim and, at the right time, shows the spoofed and malicious activity to steal credentials. This is the most used phishing variant and its effectiveness is well known. However, we recognize that other interesting variants of this attack are possible. For example, the attacker could execute her attack while the victim application is running, showing a generic error message and luring the victim to re-enter his credentials. Or, the attacker may show an error message to the victim, even when the application is not in use, in the hope that the victim enters the credentials. At the moment, our system is not able to identify and block these variants because these attack configurations are not necessarily based on polling. We note that, to date, the effectiveness of these new variants is unknown, and that it would be interesting to perform a user study.

XI. Related work

Detecting Side-Channel on Android. Several previous works focused on finding vulnerable APIs leading to state inference attacks. One such example is by Chen et al. [6], which found an information leakage exploiting the shared-memory information present in the `/proc/$PID/statm` file. Bianchi et al. [4] also found multiple leaks in the "procfs" filesystem as well as vulnerable APIs that can be used to mount state inference attacks, like `getRunningTasks`. Two more recent proposals are [12], which exploited `a11y` information leaks to also mount phishing attacks, and [11] which used the `transaction_log` of the `Binder` component to list the transactions occurring between processes. These works were mostly based on manual analysis and inspired the community to work on automatic detection of such vulnerabilities: ProcHarverster [26], SCAnDroid [27] (already discussed),and our own work.

Phishing Attack and Defense. Phishing on Android is a form of User Interface attack [10]. This paper focuses on the configuration known as "task hijacking" and it has been subject of different works. Several of them tried to identify new techniques to mount this attack, like [12] [24] and [2]. Ren et al. [23] show how it is possible to mount "task hijacking" attacks by exploiting vulnerabilities of the Android multitasking and the Activity Manager Service design. However, task hijacking is not the only available configuration: Xu et al. [34] identified how it is possible to abuse fake notifications and fake icons to lure the user into interacting with a malicious application without exploiting side-channels. Yang et al. [36], instead, exploited "Differential Context Vulnerabilies," a class of vulnerabilities and design flaws afflicting WebView, to mount phishing attack. On the defensive side, several works tried to eradicate the phishing problem on Android. Longfei et al. [33] combine several OCR techniques to detect spoofed UI and verify if the activity shown to the user is authentic or spoofed. A similar approach is shown in [19]: they introduce the "Visual Similarity Perception" technique to identify forged UI. Another work in the same category is [22]: it designs the Android Window Integrity policy system which makes sure that a sensitive activity cannot be obscured by other activities. Cooley et al. [7] instead, introduces the concept of Trusted Activity Chains to protect apps from phishing attacks by defining a sequences of activities that should not be interrupted. This sequence cannot be hijacked, otherwise a security warning will be raised. One last related work is "LeaveMeAlone" [38], already discussed in Section IX-E.

XII. Conclusion

In this work, we show how the Android platform is still affected by state inference attacks. We systematically extended the attack surface, and we designed a new automatic framework that discovered 18 new vulnerable APIs that leak sensitive information, affecting both Android 8.1 and 9. As a second contribution, we characterized polling behaviors in malicious and, more importantly, benign apps, uncovering differences that allow their proper classification. We leverage these findings to design and implement a new on-device detection mechanism that blocks state inference attacks at their root, even when exploiting unknown vulnerable APIs, with a negligible overhead, and without sacrificing usability.

Acknowledgements

We would like to thank our shepherd, Ben Andow, and the anonymous reviewers for their constructive feedback. As tradition has it, we would also like to thank Betty Sebright for her support over the past years.

References

[1] Jagdish Achara. *Unveiling and Controlling Online Tracking*. PhD thesis, 10 2016.

[2] Efthimios Alepis and Constantinos Patsakis. "Trapped by the UI: The Android Case". In Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis, editors, *"Proceedings of the International Symposium Research in Attacks, Intrusions, and Defenses (RAID)"*, pages 334–354, Cham, 2017. Springer International Publishing.

[3] AndroidRank. AndroidRank, open android market data since 2011. https://www.androidrank.org. Accessed: January 8, 2021.

[4] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. What the App is That? Deception and Countermeasures in the Android User Interface. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*, pages 931–948, Washington, DC, USA, 2015. IEEE Computer Society.

[5] Broadcom. Android malware finds new ways to derive current running tasks. https://community.broadcom.com/symantecenterprise//community-home/librarydocuments/viewdocument?DocumentKey=d3231e0f-67a0-4b31-8adb-4247ca23243dCommunityKey=1ecf5f55-9545-44d6-b0f4-4e4a7f5f5e68tab=librarydocuments. Accessed: January 8, 2021.

[6] Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. Peeking into Your App Without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, pages 1037–1052, Berkeley, CA, USA, 2014. USENIX Association.

[7] Brett Cooley, Haining Wang, and Angelos Stavrou. Activity Spoofing and Its Defense in Android Smartphones. In *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*, 2014.

[8] Corbin Davenport. Google will remove Play Store apps that use Accessibility Services for anything except helping disabled users. https://www.androidpolice.com/2017/11/12/google-will-remove-play-store-apps-use-accessibility-services-anything-except-helping-disabled-users/. Accessed: January 8, 2021.

[9] dtmilano. AndroidViewClient. https://github.com/dtmilano/AndroidViewClient. Accessed: January 8, 2021.

[10] Adrienne Porter Felt and David Wagner. Phishing on Mobile Devices. In *Proceedings of the Web 2.0 Security and Privacy*, 2011.

[11] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J. Alex Halderman, Z. Morley Mao, and Atul Prakash. "Android UI Deception Revisited: Attacks and Defenses". In Jens Grossklags and Bart Preneel, editors, *"Financial Cryptography and Data Security"*, pages 41–59, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[12] Yanick Fratantonio, Chenxiong Qian, Simon Chung, and Wenke Lee. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, May 2017.

[13] Google. UsageStatsManager Documentation. https://developer.android.com/reference//app/usage/UsageStatsManager. Accessed: January 8, 2021.

[14] Yacong Gu, Yao Cheng, Lingyun Ying, Yemian Lu, Qi Li, and Purui Su. "Exploiting Android System Services Through Bypassing Service Helpers". In Robert Deng, Jian Weng, Kui Ren, and Vinod Yegneswaran, editors, *"Security and Privacy in Communication Networks"*. Springer International Publishing, 2017.

[15] Chandraiah Jagadeesh. Red Alert 2.0: Android Trojan targets security-seekers. https://news.sophos.com/en-us/2018/07/23/red-alert-2-0-android-trojan-targets-security-seekers/. Accessed: January 8, 2021.

[16] Kaspersky. Asacub Android Trojan: From Information Stealing to Financial Fraud. https://www.kaspersky.com/about/press-releases/2016_asacub-android-trojan-from-information-stealing-to-financial-fraud. Accessed: January 8, 2021.

[17] Sun Kevin. BankBot Found on Google Play and Targets Ten New UAE Banking Apps. https://blog.trendmicro.com/trendlabs-security-intelligence/bankbot-found-google-play-targets-ten-new-uae-banking-apps/. Accessed: January 8, 2021.

[18] Nick Kralevich. Honey, I Shrunk the Attack Surface. Adventures in Android Security Hardening. https://www.blackhat.com/docs/us-17/thursday/us-17-Kralevich-Honey-I-Shrunk-The-Attack-Surface-Adventures-In-Android-Security-Hardening.pdf. Accessed: January 8, 2021.

[19] Luka Malisa, Kari Kostiainen, and Srdjan 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 (CODASPY)*, pages 289–300, New York, NY, USA, 2017. ACM.

[20] Lorenz Nicole. MysteryBot - the Android malware that's keylogger, ransomware, and trojan. https://blog.avira.com/mysterybot-the-android-malware-thats-keylogger-ransomware-and-trojan/. Accessed: January 8, 2021.

[21] Duy Phuc Pham, Croese Niels, and Han Sahin Cengiz. Exobot - Android banking Trojan on the rise. https://www.threatfabric.com/blogs/exobot_android_banking_trojan_on_the_rise.html. Accessed: January 8, 2021.

[22] Chuangang Ren, Peng Liu, and Sencun Zhu. WindowGuard: Systematic Protection of GUI Security in Android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[23] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. Towards Discovering and Understanding Task Hijacking in Android. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 945–959, Berkeley, CA, USA, 2015. USENIX Association.

[24] Hossain Shahriar, Tulin Klintic, and Victor Clincy. Mobile Phishing Attacks and Mitigation Techniques. In *Journal of Information Security*, volume 06, pages 206–212, 06 2015.

[25] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason M Ott, and Zhiyun Qian. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[26] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. Procharvester: Fully automated analysis of procfs side-channel leaks on android. In *Proceedings of the 2018 Asia Conference on Computer and Communications Security (ASIACCS)*, pages 749–763, New York, NY, USA, 2018. ACM.

[27] Raphael Spreitzer, Gerald Palfinger, and Stefan Mangard. SCAnDroid: Automated Side-Channel Analysis of Android APIs. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, pages 224–235, New York, NY, USA, 2018. ACM.

[28] ThreatFabric. Anubis II - malware and afterlife. https://www.threatfabric.com/blogs/anubis_2_malware_and_afterlife.html. Accessed: January 8, 2021.

[29] ThreatFabric. BianLian - from rags to riches, the malware dropper that had a dream. https://www.threatfabric.com/blogs/bianlian_from_rags_to_riches_-the_malware_dropper_that_had_a_dream.html. Accessed: January 8, 2021.

[30] Federico Tomassetti. JavaParser - Parser and Abstract Syntax Tree for Java. https://github.com/javaparser/javaparser. Accessed: January 8, 2021.

[31] Ventura Vitor. Gustuff banking botnet targets Australia. https://blog.talosintelligence.com/2019/04/gustuff-targets-australia.html. Accessed: January 8, 2021.

[32] Gahr Wesley, Duy Phuc Pham, and Croese Niels. LokiBot - The first hybrid Android malware. https://www.threatfabric.com/blogs/lokibot_the_first_hybrid_android_malware.html. Accessed: January 8, 2021.

[33] L. Wu, X. Du, and J. Wu. MobiFish: A lightweight anti-phishing scheme for mobile phones. In *Proceedings of the 23rd International Conference on Computer Communication and Networks (ICCCN)*, pages 1–8, Aug 2014.

[34] Zhi Xu and Sencun Zhu. Abusing Notification Services on Smartphones for Phishing and Spamming. In *Proceedings of the 6th USENIX Conference on Offensive Technologies (WOOT)*, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.

[35] Yuxuan Yan, Zhenhua Li, Qi Alfred Chen, Christo Wilson, Tianyin Xu, Ennan Zhai, Yan-Ping Li, and Yunhao Liu. Understanding and Detecting Overlay-based Android Malware at Market Scales. 2019.

[36] Guangliang Yang, Jeff Huang, and Guofei Gu. Iframes/Popups Are Dangerous in Mobile WebView: Studying and Mitigating Differential Context Vulnerabilities. In *Proceedings of the 28th USENIX Conference on Security Symposium*, 2019.

[37] Lei Zhang, Zhemin Yang, Yuyu He, Zhenyu Zhang, Zhiyun Qian, Geng Hong, Yuan Zhang, and Min Yang. Invetter: Locating Insecure Input Validations in Android Services. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[38] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave Me Alone: App-Level Protection against Runtime Information Gathering on Android. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 915–930, May 2015.

APPENDIX

A. *Case-Study*

(a) This code can be used to exploit `isAppForeground` API: the attacker only needs to provide the target UID. This API does not require permission to be invoked

```
void attack(int uid) {
  final Handler handler = new Handler();
  handler.postDelayed(new Runnable() {
  /* Executed every second */
  public void run() {
    try {
      /* Obtain a reference to IActivityManager */
      if (iam.isAppForeground(uid)) {
        /* Hijack the original activity */
      }
      handler.postDelayed(this, 1000);
    } catch (Exception e) {
        /* Handle the exception */
    }
  }, 1000);
}
```

(b) Pseudo-code to exploit `getDataLayerSnapshotForUid` API

```
/* First measure of txPackets */
public long prevTxPackets;
void attack(int uid) {
  final Handler handler = new Handler();
  handler.postDelayed(new Runnable() {
  /* Executed every second */
  public void run() {
    try {
      /* Obtain
          a reference to INetworkStatsManager */
      NetworkStats
          ns = inss.getDataLayerSnapshotForUid(uid);
      /* 1 is for foreground data
      Check if the application is sending
          data and if is trasmitting in foreground
      */
      if (ns.set
          == 1 && ns.txPackets > prevTxPackets) {
        /* Hijack the original activity */
      }
      prevTxPackets = ns.txPackets;
      handler.postDelayed(this, 1000);
    } catch (Exception e) {
        /* Handle the exception */
    }
  }, 1000);
}
```

(c) Pseudo-code to exploit `queryEvents` API

```
public long prevTime = System.currentTimeMillis();
public String TARGET_APP_PACKAGE_NAME = "com.bank"
void attack() {
  final Handler handler = new Handler();
  handler.postDelayed(new Runnable() {
  /* Executed every second */
  public void run() {
    try {
       UsageStatsManager usm
           = (UsageStatsManager)getSystemService(Context.USAGE_STATS_SERVICE);
       UsageEvents ue = usm.queryEvents(prevTime, System.currentTimeMillis());
       prevTime = System.currentTimeMillis();
       while (ue.hasNextEvent()) {
          UsageEvents.Event e = new UsageEvents.Event();
          ue.getNextEvent(e);
          if (e.getPackageName().equalsIgnoreCase(TARGET_APP_PACKAGE_NAME)) {
             if (e.getEventType() == 1) {
                /* Hijack the original activity */
             }
          }
       }
       handler.postDelayed(this, 1000);
    } catch (Exception e) {
       /* Handle the exception */
    }
  }, 1000);
}
```

B. API Whitelisting

Category	API	Example
Graphical User Interface (GUI)	For this category, we whitelist APIs from the following classes: • android.ui.ISurfaceComposer • android.gui.DisplayEventConnection • android.gui.IGraphicBufferProducer • android.gui.SensorEventConnection • android.view.IWindowSession • android.hardware.display.IDisplayManager	The GUI system handles all the operations that allows the system to display and render the UI of a given app. The application is in charge, for instance, of declaring all the supported screen sizes and pixel densities, but it does not have to handle the interaction with the actual frame buffer. The GUI framework will handle, behind the scene all the rendering operations and the rescaling, if needed.
Audio and Video	For this category, we whitelist mostly APIs from the *android.media* package. This package, provides classes that manage various media interfaces in audio and video. For instance, we whitelist: • android.media.IMediaAnalyticsService • android.media.IMediaCodecService • android.media.IMediaExtractorService • android.media.IMediaMetadataRetriever • android.media.IMediaRouterService • android.media.IMediaPlayerService • android.media.IAudioService • android.media.IAudioPolicyService	The Audio and Video services on Android is a complex ecosystem formed of different components. Every component is in charge of a specific task. For instance, when an application wants to play an audio, it normally relies on the "MediaPlayer" component, and performs operations like "start, stop, and pause." However, behind the scenes, all the whitelisted components performs the tasks of handling the Audio, using the correct Decoder and Coded, forward the audio to the proper hardware interface and handle the refresh of the audio buffer.
Digital Rights Management (DRM)	For this category, the whitelist contains the classes of the *drm* package, which handles all the DRM framework.	DRM is a complex framework: it relies on plugins and it is strictly connected with the "Media" system. In fact, DRM content are normally audio and video file, protected with digital rights, that are played by the system player's. For example, every time the app starts the DRM, a series of operations are done behind the scenes, like loading different DRM Plugins, setup the connections with MediaPlayer and the Media System, to finally decodes and then forwards to the player each chunk of the file to play.
System Services Internals	This categories contains a variety of API that are used by the system, behind the scenes, when dealing with different system components. For instance, the system automatically handles from the "synchronization" operations for what concerns the access to shared structures to the "reference counting" when dealing with Content Providers. We whitelist APIs for the following services: • ContentProvider • PowerManager • PermissionManager • AlarmManager	As mentioned before, when an application use a system services shared accross multiple apps, it does not have to handle all the operation to acquire and release the lock. In fact, we noticed these operations are handled directly by the service on behalf of the app. It is possible to see the same behavior when dealing with reference counting, for example when interacting with ContentProviders or other components that can be shared across multiple apps. Some of the APIs that we identified are used by the system services to achieve these tasks are `acquireWakeLock`, `releaseWakeLock`, or `refContentProvider`

TABLE V: The table summarizes and shows the different categories of APIs we whitelist on our on-device defense system. For each category, we describe the classes, services, APIs, or packages we whitelist and we provide a detailed description with concrete example. We manually investigated each of the APIs in our whitelist and none of these APIs can abused by malicious apps to mount state inference attacks.