2015 IEEE Symposium on Security and Privacy

Leave Me Alone: App-level Protection Against Runtime Information Gathering on Android

Nan Zhang*, Kan Yuan*, Muhammad Naveed†, Xiaoyong Zhou* and XiaoFeng Wang*
*Indiana University, Bloomington
Email: {nz3, kanyuan, zhou, xw7}@indiana.edu
†University of Illinois at Urbana-Champaign
Email: naveed2@illinois.edu

***Abstract*—Stealing of sensitive information from apps is always considered to be one of the most critical threats to Android security. Recent studies show that this can happen even to the apps without explicit implementation flaws, through exploiting some design weaknesses of the operating system, e.g., shared communication channels such as Bluetooth, and side channels such as memory and network-data usages. In all these attacks, a malicious app needs to run side-by-side with the target app (the victim) to collect its runtime information. Examples include recording phone conversations from the phone app, gathering WebMD's data usages to infer the disease condition the user looks at, etc. This *runtime-information-gathering* (RIG) threat is realistic and serious, as demonstrated by prior research and our new findings, which reveal that the malware monitoring popular Android-based home security systems can figure out when the house is empty and the user is not looking at surveillance cameras, and even turn off the alarm delivered to her phone.**

To defend against this new category of attacks, we propose a novel technique that changes neither the operating system nor the target apps, and provides immediate protection as soon as an ordinary app (with only normal and dangerous permissions) is installed. This new approach, called *App Guardian*, thwarts a malicious app's runtime monitoring attempt by pausing all suspicious background processes when the target app (called *principal*) is running in the foreground, and resuming them after the app stops and its runtime environment is cleaned up. Our technique leverages a unique feature of Android, on which third-party apps running in the background are often considered to be disposable and can be stopped anytime with only a minor performance and utility implication. We further limit such an impact by only focusing on a small set of *suspicious* background apps, which are identified by their behaviors inferred from their side channels (e.g., thread names, CPU scheduling and kernel time). App Guardian is also carefully designed to choose the right moments to start and end the protection procedure, and effectively protect itself against malicious apps. Our experimental studies show that this new technique defeated all known RIG attacks, with small impacts on the utility of legitimate apps and the performance of the OS. Most importantly, the idea underlying our approach, including app-level protection, side-channel based defense and lightweight response, not only significantly raises the bar for the RIG attacks and the research on this subject but can also inspire the follow-up effort on new detection systems practically deployable in the fragmented Android ecosystem.

I. Introduction

The popularity of Android devices comes with a vibrant application (*app* in short) market. New apps continue to emerge, providing services ranging from news, weather, and entertainment to such serious businesses as banking, medical, finance, and even home security. Apps for these businesses carry sensitive personal information (e.g., bank account details, diseases and medicines, investment secret, etc.) that needs to be protected from unauthorized programs running on the same device. Serving this purpose is the Android security model that confines each app within its *application sandbox* using a unique Linux user ID to prevent it from accessing other apps' data. In spite of the protection in place, through shared communication channels (e.g., audio, Bluetooth) or public resources (e.g., memory, CPU usage), sensitive user data could still be disclosed to the malicious app that continuously monitors the victim app's activities and collects its runtime information from those sources. Such *runtime information gathering* is known to be one of the most serious threats to Android users' privacy, as extensively reported by prior studies [1]–[7].

Runtime information gathering. More specifically, "runtime information gathering" (*RIG*) here refers to any malicious activities that involve collecting the data produced or received by an app during its execution, in an attempt to directly steal or indirectly infer sensitive user information. Such an attack can happen by abusing the permission the malicious app acquired from the user, e.g., unauthorized recording of the user's phone conversation, or through analyzing a set of *side-channel* information disclosed by the app, e.g., its CPU, memory and mobile-data usages [1], [5]. For example, prior research shows that apps with the `RECORD_AUDIO` permission are capable of selectively extracting confidential data (e.g., credit card number) and stealthily delivering it to the adversary [3]. Also, the official app of an external medical device, such as a blood glucose meter, can be monitored for collecting patient data from the device through the Bluetooth channel, before the official app is able to establish its connection with the device [6]. Particularly concerning here is that even the app *not asking for any permission* can still obtain highly-sensitive user information from a variety of side channels, demonstrating the fundamental weakness of mobile devices in separating an app's operations from its data. Examples include web content detected through analyzing the browser's memory footprints [5], key strokes logged using the phone's accelerometer [4] and the mobile user's identity, disease and financial information inferred from different apps' mobile-data usages [1].

In addition to those known instances of the RIG threat, we further looked into its implication to Android controlled *Internet of Things* (IoT), which are emerging systems increasingly used for smart home [8], [9], automobile control [10], [11], home security [12], etc. The first step we took includes an analysis of

DOI 10.1109/SP.2015.61

two highly popular home-security IoT systems, Belkin NetCam Wi-Fi Camera with Night Vision [13] and Nest Protect [9], both of which have been extensively used [14]. Our preliminary study shows that they are all vulnerable to the RIG threat. For example, we found that through the official app of NetCam, a malicious app without permission can find out when no one is at home and the phone user is not looking at the surveillance video (through the official app on her phone); also it knows when the camera's motion sensor captures the presence of a stranger at home and is sending an alarm message to the user's phone, which enables the malware to turn off the phone's speaker, making the alarm go unnoticed. This actually helps a robber break into one's home without being discovered, even when the home is protected by such a security system. A demo of the attack is posted online [15]. These findings, together with what are reported in the prior research, point to the urgent need to mitigate the RIG threat to mobile devices.

Challenges. Conventional solutions to the problem rely on modifying either the Android OS or the apps under the threat. Specifically, one can enhance Android's access control mechanism to prevent information leaks during security-critical operations such as phone calls, and remove the public resources that could be used for a side-channel analysis. This, however, inevitably makes the system less usable and causes compatibility issues for the apps that already utilize the public resources for legitimate purposes (mobile-data monitor [16]). Most importantly, due to the fragmentation of the Android ecosystem, deployment of any OS-level solution is often complicated and painful: whenever Google comes up with a patch, individual device manufacturers need to customize it for all their devices before passing its variations to the carriers, who will ultimately decide whether to release them and when to do that. Even in the case that the manufacturers are willing to build the protection into their new products, given the slow pace with which Android devices are upgraded, it is almost certain that the new protection will take a long time before it can reach any significant portion of the 1 billion Android devices worldwide. On the other hand, new RIG attacks continue to be brought to the spotlight [2], [4], [5], [7], [17]–[19], effective mitigation is therefore in an urgent need for safeguarding Android users' private information. Furthermore, pushing the problem to the app developers is by no means a good idea, as it is less clear what an app can do by itself to control its information exposed by the OS: for example, it cannot disable the recording activity of another app; also adding noise to an app's CPU, memory and data statistics may not eliminate the side-channel leaks and certainly increases its performance overhead.

App Guardian. In our research, we found that the RIG attacks can be defeated on the application level, *without touching the OS or the apps under protection at all*. What we come up with is just an ordinary app, called *App Guardian* or simply *Guardian*, that can be posted on the Google Play store and installed by any Android user on her device to acquire immediate protection of her security-critical apps. This is achieved, in a nutshell, by pausing all background apps capable of causing damage to the information assets of the protected app (called *principal* in our research) when it is running in the foreground, and resuming those apps (as they might not be actually malicious) after the principal finishes its tasks and its data (e.g., process files and caches) has been sanitized. Without access to the principal's runtime information, a RIG attempt (no matter what channel it is aimed at) can no longer be successful.

More specifically, on an unrooted phone, the pause/resume operations are performed through closing suspicious apps and later restarting them, using the relevant dangerous level permission. Due to the unique feature of the Android OS, which allows most third-party apps running in the background to be terminated when the memory runs low and also provides the mechanism to preserve their states, this approach has only a limited impact on those apps' legitimate operations. The impact becomes even less significant with a strategic selection of only a small set of suspicious apps to stop, based upon their observable features. Our Guardian app has been carefully designed to determine when to put the protection in place and when to lift it, after properly cleansing the principal's public resources of sensitive data. Most importantly, it has been built to protect itself against the attacks from malicious apps and defeat different tricks (such as collusion) they play.

A unique feature of Guardian is its strategy to identify suspicious apps. This is done by inspecting individual apps' permissions and *behaviors*. Note that finding such behaviors is nontrivial for a non-system app like Guardian, since it cannot see the system-call level activities of other apps. In our research, we developed a new technique that *leverages an app's side-channel information to infer its activities*. Such information includes a set of public data, such as the name of a service thread, a thread's scheduling status and the amount of kernel time it consumes. For example, an untrusted app can get caught when it is trying to record a phone conversation, once Guardian observes that the Audio service process spawns a new thread called `AudioIn_X` (indicating a recording activity) and the suspicious app (with the `RECORD_AUDIO` permission) utilizes CPU. Also, a third-party background process, unrelated to the principal, could look risky to the principal if it is frequently scheduled to use CPU, as the CPU cycles here could be spent on RIG. Using such side-channel information, Guardian carefully chooses the targets to close, to minimize the utility and performance impacts of the operation without undermining the security protection.

We implemented App Guardian and evaluated its utility over 475 most popular apps in 27 categories on Google Play. We found that under the strategy for selecting suspicious processes, only 1.68% of the popular apps with perceptible impacts on user experience needed to be closed when they were running in the background and all of them could be swiftly restored without losing their runtime states. Our study further shows that the new technique defeated all known RIG attacks, including audio recording, Bluetooth misbonding [6], a series of side-channel attacks on high-profile apps [1], [4], [5], [20], the recently proposed user-interface inference [2] and voice eavesdropping [7], together with the new IoT attacks we discovered, at a performance cost as low as 5% of CPU time and 40 MB memory.

Contributions. The scientific contributions of the paper are outlined as follows:

• *New understanding of the RIG threat*. We investigated the RIG threat to Android-controlled IoT systems, which reveals serious side-channel leaks from popular IoT systems (e.g., disclosing when one's home is empty).

• *New protection and new bar for the RIG research*. We

developed a novel application-level defense against the RIG threat, which has been built into an ordinary app and can therefore be easily distributed to a large number of Android devices to provide immediate protection. More importantly, given its promise of a real-world deployment and the technique's effectiveness against the known attacks, not only does our approach make real-world RIG exploits more difficult to succeed, but it has also noticeably raised the bar for the scientific research in this active area [2], [4], [5], [7], [17]–[19]: now new attacks discovered will be put to the test of our defense, to make the case that they indeed pose a realistic threat. This will certainly move the security research in this domain forward.

• *Novel side-channel based detection and lightweight response*. Up to our knowledge, we are the first party that leverages side channels to detect side-channel attacks and other malicious activities on mobile devices. Our unique observation is that on these devices, a malicious app needs to aggressively utilize CPU and other computing resources to gather useful information from a target app during its runtime. Such behavior can actually be observed from the attacker's own side channels, allowing a third-party detection system to discover the attack without access to system-level information (e.g., the attacker's API calls). This effort is further supported by a lightweight response to the suspicious activities identified, which just temporarily suspends a suspicious app's operation when important things are happening, and resumes it later. The cost for a false alarm is therefore minimized. Such an idea could find its way to apply to other security domains, inspiring follow-up research on app-level intrusion detection on mobile systems.

• *Implementation and evaluation*. We implemented our design and tested it on 475 popular apps. Our evaluation demonstrates the efficacy of our new technique.

Roadmap. The rest of the paper is organized as follows: Section II introduces the RIG threat to mobile devices and elaborates our new study on its implications to Android IoT; Section III describes our design and implementation of App Guardian; Section IV reports our evaluation study on the new technique; Section V discusses the limitations of our current approach and potential future research; Section VI reviews related prior work and Section VII concludes the paper.

II. Menace of Runtime Information Gathering

As discussed before, runtime information gathering poses a serious threat to Android user's privacy. In this section, we introduce background information about Android security and prior studies on this problem. Then, we report our preliminary investigation on two popular Android home security systems, whose sensitive information (e.g., whether a house is empty) can be recovered by RIG attacks.

A. Background and Prior Findings

Following we describe how Android protects its apps, and why such protection is insufficient to stop RIG attacks.

Android security and RIG. Android security model is characterized by its unique application sandbox, which has been built on top of Linux's kernel-level protection (e.g., process separation, file system access control). Specifically, each app is assigned a unique Linux user ID (UID), which separates it from other apps. As a result, except for a set of shared resources the app utilizes and its runtime statistics made public by the OS, e.g., virtual files under the process file system (*proc*), its operations and data are beyond other apps' reach. To use protected global resources, such as audio, video, GPS, etc., each app needs to request permissions from the user or the OS before its installation. Such permissions are categorized into different protection levels [21], among which *normal* ones are automatically granted to the apps when asked, *dangerous* permissions are given based upon the user's consent, and *system* or *signature* permissions are saved for system apps. With a proper permission, an app can call relevant APIs to operate on those global resources, e.g., recording audio, taking pictures, connecting to Bluetooth accessories, etc.

This security model is known to have a few issues, which are becoming prominent in the presence of increasingly diverse Android applications. First, the permission-based access control turns out to be too coarse-grained: any app granted a permission is allowed to use it to access any resources, under any circumstances. For example, a voice recorder can tape any phone conversation without restriction; a game app with the Bluetooth permission for connecting to its playpad can also download patient data from a Bluetooth glucose meter. Further, the model does not protect an app's runtime statistics and other resources the OS considers to be public. An example is its network-data usage. Under some circumstances, such information could actually be linked to the app's program states, allowing the adversary to figure out the content of its data [1]. As a consequence of these design limitations, even a carefully-implemented app often unwittingly discloses its confidential data through the way it uses resources (CPU, memory, network data, etc.) during its execution or through shared communication channels (audio, Bluetooth, etc.) when it is sending or receiving the data. This subjects the app to all kinds of RIG attacks in which the adversary is continuously monitoring its operations and collecting its runtime information.

Data stealing. Specifically, unauthorized voice recording has long been known to be a serious security issue. Prior study shows that malware recording phone conversations can masquerade as an app with a legitimate need for the related permission, such as a voice dialer or a voice memo application [3]. Once installed, it can be made to intelligently choose the data of a high value (e.g., credit card number, password) to steal, leveraging context information such as a bank's interactive voice response system. In such an attack, the malware operates when the system phone app runs in the foreground to command Android's MediaRecorder service to collect the voice data exchanged during a phone call.

More recently, research has found that Android Bluetooth accessories are also vulnerable to such runtime data stealing [6]. The official app of a Bluetooth medical device, such as blood-glucose meter and pulse oximeter, can be monitored by a malicious app with the Bluetooth permission. Once the legitimate app starts running in the foreground, the malware tries to connect to its accessory before the app does or right after it finishes its communication but before the device is turned off. This RIG attempt was found to be often successful, letting the unauthorized app download a patient's clinic data. Another example is the attack on programmatic screenshot apps [20], which typically run a local socket connection to command a

process invoked through Android Debug Bridge (ADB), so as to get ADB's signature permission for screenshot taking. The problem is that this local socket channel has not been properly regulated and as a result, any app with the Network permission can ask the process to snap an picture of the screen. The research shows that using this technique, a RIG attack can continuously take screenshots when the user types into an app, extracting the sensitive information (e.g., password) she enters.

In all those attacks, a malicious app abuses permissions it gets to directly collect sensitive user data from the target app running in the foreground. Following we show that even in the absence of such permissions, RIG attacks can still happen through a variety of side channels on Android.

Side-channel inference. Android is designed for thin devices, on which the level of concurrency is limited: typically, the foreground process controls most resources while those running in the background are often inactive and considered to be disposable. Also, most apps are just the user interfaces of web applications, and characterized by simple designs and intensive interactions with their web services during their operations. These features make an Android app's behavior conspicuous to the party continuously monitoring its CPU, memory, network-data usages and other side channels and vulnerable to the inference attacks that link the information collected to the content of its data such as the user's inputs.

Specifically, prior research studied the RIG attacks through the side channels on both the Linux layer and Android's application framework layer. Linux-level channels are mostly related to the public process filesystem, which includes public statistics about a process's use of memory, CPU, network data and others. For example, the dynamics of the browser's memory usages (observed from `/proc/<pid>/statm`) during its rendering of web content are found to be useful for identifying the web page the user visits [5]. In this attack, a malicious app continuously samples the browser's data resident sizes (called *memory footprint*) when it is loading a page and compares the set of the footprints with the profiles of web sites. A more recent study looked into Android's network data usages (`/proc/uid_stat/tcp_snd` and `tcp_rcv`), and shows that the increments observed from these two indicators are in line with the payload sizes of the TCP packets sent or received by the app under the surveillance. Such increments were used to fingerprint the app's activities, such as sending a tweet. This allows the adversary to query the Twitter server using the timestamps of such operations, for determining the individual who tweets at all these times. Also such usage increments were found to be sufficient for identifying the content the user clicks on when using the most popular healthcare app WebMD [22] and high-profile investment app Yahoo! Finance [23]. As a result, by simply examining the increment sequences gathered from these apps' runtime, a malicious app without any permission can figure out the disease and stock a user is interested in.

On the framework layer, what have been extensively investigated include keystroke identification using motion sensors such as accelerometer. The idea is to monitor the movement and gesture changes when the user types through the touch screen to infer the content she enters into a running app. Such an inference was found to be completely feasible [4]. Another side channel exploited on the framework layer is the public API function. Particularly, prior research shows that an Android user's driving route could be determined by looking at sequences of the duration for the voice guidance produced by Google Navigator. Such a duration is identified from the speaker's status ("on" or "off"), which can be found out through a public API `isMusicActive`. A collection of the duration sequences turns out to be sufficiently informative for tracking the path the user drives down [1].

B. Preliminary Study on Android IoT

Given the unique features of Android (i.e., simple user-interface programs, little noise in their running environments), we strongly believe that what have been discovered is just a tip of the iceberg. As a baby step towards a better understanding of the RIG threat to Android, we looked into two popular IoT systems, the Belkin NetCam Wi-Fi camera with Night Vision [13] and Nest Protect [9]. Both systems are among the front runners of the current trend of Android-based home security and safety IoTs [24]. The NetCam camera is designed for home surveillance and motion detection, which can identify the stranger who gets into the house and allows the house owner to check what has happened remotely, through her smartphone. Nest Protect is an intelligent fire alarm system. It monitors the fire situation in the user's home and alerts her through phone. Both systems are considered to be high-end IoTs, with at least hundreds of thousands of users [14]. Here we describe how they work and our analysis that reveals their RIG weaknesses.

The IoT systems. The way the NetCam system works is illustrated in Figure 1, which is also typical for other smartphone-based IoT systems, including Nest Protect. Specifically, such a system deploys a single or multiple sensors in the user's house. Each sensor is connected to a home Wi-Fi router for communicating with other sensors and a server operated by the party providing the service (home security or fire alarm). Whenever a situation is found, the sensor reports to the server, which takes measures to respond to the event, including pushing a message through Google-Cloud Messaging (GCM) to the user's phone. This message is picked up by the GCM process on the phone, which forwards it to the IoT's official app through an Intent. The app further posts a notification, producing an alert sound to arouse the user's attention. The official app also enables the user to remotely control the sensors and check the data (e.g., looking at a live video) they collect.

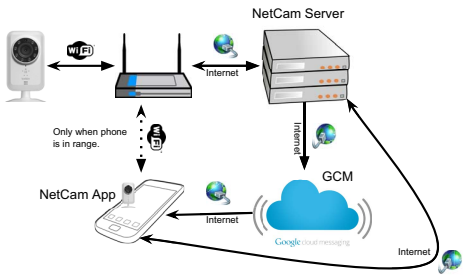


Fig. 1: NetCam system

For the NetCam system, whenever the user is leaving home, she can turn on the camera's motion detector by clicking on the official app's "save clips" switch. Once the status of this switch changes, the app communicates with the server to configure the camera to automatically identify the motion likely related

to human activities (which is not supposed to happen, given that the house is empty), and reports to the user through the GCM channel. Similarly, Nest Protect sensors capture a fire situation and sends an alarm to the user's phone.

Analysis and findings. In our research, we ran both systems while monitoring their operations through an unprivileged attack app on the user's Android phone. The app utilizes `getRunningTasks` to find out whether its target starts running, and continuously collects the target's side-channel information, particularly its network data usage (`tcp_snd` and `tcp_rcv`) and CPU usage (`/proc/<pid>/stat`), for inferring the events related to the target. With such information, the app can also actively interfere with the system's operation, to prevent the user from being properly notified.

We found that for NetCam, the status of the "save clips" switch is actually observable from the official app's network data usage. This is important because when the switch is on (means that the camera reports *any* human-related motions detected), we know exactly that no one is at home. Specifically, whenever the status changes, the official app always sends out a single packet with a payload of $368 + 2k$ bytes, where k is the length of one's username, which is typically around 10 bytes. This packet causes the app's `tcp_snd` to rise by its payload size while a response package may add to `tcp_rcv`, and then remains unchanged, together with `tcp_snd` in at least 1 seconds. These features make the switch-setting operations (turning it on or off) stand out, as no other activities involve a single packet with that length (above 300 bytes). Also, when we take a close look at the packets that change the switch status (from "off" to "on" or vice versa), the former is always one byte below the latter, even when usernames vary in length. To detect the former, our attack app first identifies a few switch operations based on their unique features (changes of `tcp_snd` and `tcp_rcv`), and then compares the exact `tcp_snd` increments they cause to find out the one that sets the switch on (one byte less than those deactivating the detector). Such information was accurately collected in our experiment when our app read from the proc file at 10 times per second. In this way, the app was able to determine exactly when the user's house is empty.

Further, our research shows that the message the GCM process delivers to the official NetCam app can also be fingerprinted. This is important because now the attack app can find out whether the camera indeed gets something and respond to such an event, for example, by muting the speaker temporarily to make the event less likely to come to the user's immediate attention. Specifically, we found that the GCM message for such an alarm (sent by the NetCam server) ranges from 266 bytes to more than 300, depending on the length of the username and other variables. Again, this can be seen from the increment of `tcp_rcv` (for the GCM process). This length is rare for messages processed by GCM but might not be unique, given that any app can use this channel to get messages. Therefore, our attack app further verifies the recipient of the message by looking at whether the official app of NetCam is invoked or its background process starts using CPU resources (`/proc/<pid>/schedstat`) right after the arrival of the message (in 150 milliseconds). If so, it is evident that the alarm has come. In response to it, the attack app immediately turns off sound and vibration, and restores the original settings after 10 seconds. To make the attack succeed, the app needs to check the GCM's proc files at 20 times per second.

Another trick we can play is to find out whether the user is looking at the live video streamed from the camera. A unique feature for this operation was found to be the arrival of 6 consecutive packets, each with at least 2,500 bytes. This can be observed from the increments of the NetCam app's `tcp_rcv` when the attack app collects the data 5 times per second. Putting things together, we conclude that even though the IoT system is for home security, its side-channel weaknesses can actually be taken advantage of for committing a robbery. Specifically, the robber running an app on the victim's phone knows when the house is empty by inferring the switch status, whether the camera detects his break-in and the user is looking at the video. He can be further protected by muting the alarm sent to the user. A video demo of the attack is posted online [15].

It turns out that Nest Protect is equally vulnerable to the RIG attacks, though the system was carefully built to avoid common security flaws[1]. Specifically, the fire alarm sent through GCM always increases its `tcp_rcv` by 305 to 318 bytes, which can be reliably identified by the attack app when it is sampling the indicator 20 times per second. The event can be confirmed by checking the CPU usage of the Nest app. In our research, we performed the same muting attack to disable sound once an alarm is arrived, which worked as effectively as that on NetCam. As a result, the attack app could make the alarm temporarily go unnoticeable.

III. App Guardian

As demonstrated by the prior research and our preliminary study, Android is not designed to withstand the RIG threat. Its fundamental limitations, such as shared channels and public resources, subject it to various forms of runtime information collection, which often causes the exposure of sensitive user information. This problem is realistic, pervasive and serious, and can only be addressed by new techniques that are effective and also easy to deploy across nearly one billion Android systems customized by various parties. In this section, we elaborate the design and implementation of such a technique, which protects the app carrying private user information at the application level.

A. Overview

Before delving into details, here we first present the idea behind our technique, a 1000-foot view of its design and the assumptions made in our research.

Idea and high-level design. Critical to the success of any RIG attack is a malicious app's capability to run side-by-side with the target app, collecting the information exposed during its operation. To defeat such an attack, therefore, it is important to stop such information-gathering activities. For this purpose, our approach suspends suspicious apps' executions throughout the target app's runtime and resumes them after the target completes its task. During this period (called the *Ward* mode or simply *Ward* in our research), we further ensure that no suspicious app is invoked, and the Guardian app can protect

[1]For example, compared with other IoT systems, Nest includes carefully-designed protocols to ensure security during the communication among different sensors and between sensors and the server.

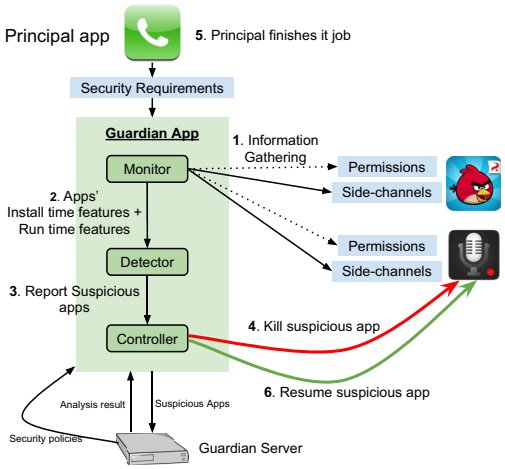


Fig. 2: Architecture of App Guardian

itself from other apps. Also such protection should not rely on any change to the Android OS or the app being protected (i.e., the principal). This is important for practical adoption of the new technique. Finally, the overhead of the protection, in terms of its impact on system performance and other apps' utility, should be minimized, which can be achieved through analyzing the behaviors of apps to close only those indeed suspicious.

This idea has been applied to build our App Guardian, an app with only normal and dangerous permissions. Its design is illustrated in Figure 2. Specifically, the app includes three key components, an *app monitor*, a *suspicious app detector* and an *app controller*. The monitor collects the features (e.g., permissions) of all third-party apps installed on the devices and during their runtime, keeps a close eye on their behaviors through periodically sampling those apps' CPU consumption and other observable behavior patterns from their side channels. Such behavior information, together with individual apps' features, is passed to the detector for identifying those that act dangerously according to the security requirements of the protected app (the principal) and a set of security policies. The suspicious apps reported are then suspended by the controller before and throughout the Ward mode, and resumed afterwards. The controller is also responsible for the safety of Guardian itself. These components can stand on their own, providing protection to the mobile user, and in the meantime, they can also be supported by a server that maintains security policies for protecting different principals against various threats, and also analyzes the apps with strong evidence to be illicit under the device user's consent.

For example, consider a hypothetical medical app that connects to a health device through Bluetooth to collect health data from a user, and also provides her information about disease conditions according to the data. Here the principal (the medical app) needs to be protected on two fronts: the Bluetooth channel it uses to download data from the device and its network-data usage that can be exploited to infer the conditions the user checks. To this end, the Guardian app monitors all other third-party apps running in the background. Once the principal is activated, Guardian closes the third-party processes that look dangerous to the principal (Section III-B), particularly, those being scheduled in the background with a high frequency. This is because if such a process was actually sampling the principal's network-data usage at this rate, it would be able to infer the condition the user is looking at (Section III-C). Particularly, the apps that change their scheduling rates, apparently based upon whether the principal is running are considered to be highly suspicious.

Further, whenever a background app with the Bluetooth permission is found to consume CPU resources, Guardian checks whether the Bluetooth service is also active (using CPU): in this case, the app needs to be stopped too, to protect the principal's data on its Bluetooth device (Section III-C). Once the principal is switched to the background or other exit (from the Ward mode) conditions are met, the Guardian app terminates the principal's process and cleans up the cache. After that, it restores closed third-party apps that need to be resumed (Section III-B).

In this way, our approach ensures that a RIG app does not get a chance to collect information from the principal when it is running in the Ward mode.

Adversary model. The Guardian app can be downloaded from an app store to provide the device user immediate protection of her security-critical apps. To make this happen, the user needs to grant Guardian a set of permissions, including `KILL_BACKGROUND_PROCESSES` for closing other third-party apps, `SYSTEM_ALERT_WINDOW` for popping up an alert to the user, `INTERNET` for Internet access, `GET_TASK` for getting top activity and `BIND_NOTIFICATION_LISTENER_SERVICE` for controlling notifications. Also, we only consider the malicious apps running in the user mode, without any system privileges, as most real-world Android malware does. Such apps have to utilize Android shared resources to steal or infer sensitive information within the protected app. For those with system privileges, however, they could break Android's application sandbox and circumvent an app-level protection.

B. Safeguarding App at Runtime

At the center of our App Guardian system is suspension and resumption of suspicious apps, which protects the principal in the Ward mode. Simple as it appears to be, the approach actually needs to be carefully designed to address a few technical challenges, e.g., when to enter the Ward (i.e, to start protection) and when to leave, how to protect the Guardian app itself, etc. Here we elaborate how our technique works, and its impact on the utility of legitimate third-party apps.

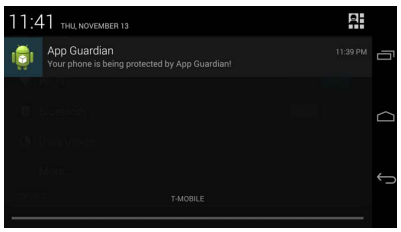


Fig. 3: Non-clearable notification of App Guardian

Self protection. Running as an ordinary third-party app, App Guardian works against the malicious apps that operate on the same OS level. Most important here is how to protect itself against the malware's attempt to terminate it. For this purpose,

we built Guardian in a way that it cannot be killed by any third-party app. Specifically, the service of our app is invoked through `startForeground`, the API that puts the service in the perceptible state, though all the user can see is just a notification posted on the Notification Center of her device (Figure 3). In this state, the app cannot be stopped by another app using the `KILL_BACKGROUND_PROCESSES` permission. Also, it will not be killed when the system is low on memory, unless under the extreme condition where the system kills all apps except system processes and those running in the foreground to free off memory [25]. Note that in practice, it is very difficult for a single app to deplete the memory on an Android device even in the foreground. Specifically, each device has a limit on how much memory a foreground app can use. The baseline specified by Android is just 16 MB [26]. In our research, we analyzed Nexus 5 (with 2 GB memory) and found that this limit has been raised to 192 MB. In a very rare case, an app could specify `largeHeap = "true"` in its manifest file to ask for maximal 512 MB, still well below what the OS can offer. Also, this unusual requirement from an untrusted third party app could raise suspicion. Even in the case when the Guardian app is indeed about to be killed, its controller will automatically generate a *restart* intent, so it is immediately revived after being stopped.

A malicious app may try to play the same trick to prevent itself from being stopped. The problem is that this attempt is highly prominent and extremely rare among legitimate apps except those with special needs (like Guardian). In our research, we inspected 475 most popular apps collected from Google Play and found that only 7, about 1.5%, have this capability. Among them are 3 launchers (home apps), 2 weather apps and 2 social-networking apps. If these "untouchable" apps are not trusted and also found to behave suspiciously, Guardian will report it to the user, asking her to stop the service before running her protected apps.

Monitoring. Once installed, Guardian's monitor module first scans all existing third-party apps, collecting their information (e.g., package names, permissions, etc.), to find out who those apps are and what they are capable of. This also happens whenever a new app is installed, which Guardian is notified (by registering a broadcast receiver and using the Intent filter `action.PACKAGE_ADDED`) and acts on to check the new app's features. Using such information, together with the security requirements from the principal (Section III-C), Guardian determines the way each app should be treated. For example, those on a whitelist are trusted and will not be tracked while the others, particularly the ones with the privileges that potentially can do harm to user privacy (e.g., the `RECORD_AUDIO` permission), will be monitored closely during their execution. 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.

During its operation, Guardian keeps a close eye on other running apps and continuously assesses their potential threats to the principal before and during the app's execution. Specifically, we implemented Guardian as a hybrid app, with its monitor component built with C++ to achieve a high runtime performance. The monitor continuously inspects untrusted apps' proc files (once per second in our experiment). For this purpose, the processes of these apps need to be identified, which is achieved as follows. We convert the process identifiers (PID) of all running processes into their user identifiers (UID) using `stat()`, and then remove all system processes (with UIDs below 10000). Those system apps are only examined by our app when they operate together with a third-party app under surveillance: for example, running an audio recording thread on behalf of the app (Section III-C). Further, the app name of each process is found from `/proc/<pid>/cmdline` and for the non-system apps, their names are used to check against the whitelist to find out those untrusted. After that, Guardian works concurrently on all untrusted processes, generating a thread for each of them.

Entering the Ward. Besides untrusted apps, Guardian also continuously monitors other system activities related to the app it protects (the principal) and initiates the whole protection procedure once an "entry" condition is met for the Ward mode, where the principal is isolated from untrusted processes. A typical entry condition is when the principal starts running in the foreground. This event is detected by the monitor that keeps track of all newly created processes through periodically running `getRunningTasks`. As soon as this happens, Guardian immediately utilizes its controller to pause all suspicious background apps. Those apps are identified by the detector according to their behaviors observed from their side channels and other conditions, which are elaborated in Section III-C. The idea here is to temporarily stop them to create a safe environment for the principal to run, and restore them afterwards. The life cycle of this protection procedure is illustrated in Figure 4.

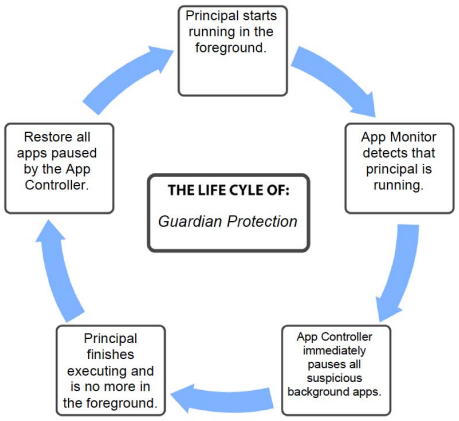


Fig. 4: The lifecycle of Guardian protection

Suspension and restoration can be easily done on a rooted phone, when Guardian has a root privilege. In this case, simple commands `kill -STOP <pid>` for pausing a process, and `kill -CONT <pid>` for resuming the process, will do the trick. However, most devices are not rooted and the Guardian app will have nothing but ordinary app's privilege to support its mission. In this case, all we can do is just to close the whole app package (which may include multiple processes). This is done through `killBackgroundProcesses(PackageName)`, when the caller of the function has the `KILL_BACKGROUND_PROCESSES` permission (at the dangerous level). This operation is unique since on Linux, a user can only close her own process, not other users'. On Android, however, once an app is switched to the background, oftentimes, it

is considered disposable, and can be terminated anytime when the system's memory runs low. Also, Android provides a mechanism to let the app to be stopped save its runtime state (`onSaveInstanceState`) and restore the state once it is launched again (`onRestoreInstanceState`). Therefore, the impact of this approach on legitimate apps is small, as demonstrated by a study elaborated later in this section.

Specifically, Android assigns each background process an `oom_adj` score, which is used to determine which background process to kill when the memory runs low. The score ranges from -17 to 15. The higher the score, more likely the process having the score is to be terminated. An app, once switched to the background, its processes typically get 9, which is given by Android to the programs that "can be killed without disruption" [27]. For the third-party app that provides a persistent background service, its `oom_adj` score should be 5 or less. We found that almost all such background processes can be terminated, and all of them keep their states. An exception is those given a score 2, the privilege level Guardian itself gets. The apps running at this level are considered perceptible to the user[2], and therefore, can only be terminated by the user. Only a few legitimate apps acquire this score, 4.42% as we found in our research (Section IV-B), and many of them need to be trusted by the user, e.g., keyboards (otherwise, the app can log all the user's inputs). The rest are the 7 apps using foreground services, as discussed before, and media players that only operate with this score when music is on. For such an app, Guardian first pauses the music and waits for the player's score to go up (and its privilege to go down). Then, if the app still looks suspicious (Section III-C), it can be suspended just like other background processes. Specifically, our approach simulates a user click on the media button: it first broadcasts an Intent `ACTION_MEDIA_BUTTON` with action `ACTION_DOWN` and key code `KEYCODE_MEDIA_PLAY_PAUSE`, and then sends another one in 50 ms with action `ACTION_UP`. At this point, music stops, the player loses its privilege and can be terminated at anytime.

As discussed before, a typical condition for entering this Ward is just the launch of the principal. Actually, the user of Guardian can also specify other conditions to trigger the whole protection procedure before or after the principal runs in the foreground. For example, our app can also monitor the GCM process. Whenever a message comes, apparently related to the principal according to its feature (i.e., the sizes of the increments for `tcp_snd` and `tcp_rcv`), Guardian can immediately stop untrusted processes to prevent them from observing the invocation of the principal. This could make the adversary difficult to determine the arrival of an event (a fire alarm), simply because it cannot confirm that this happens from the principal's operation (Section II-B). Alternatively, Guardian can register with the principal's notification event, using the `BIND_NOTIFICATION_LISTENER_SERVICE` permission, to find out when the protected app posts a notification. Once this happens, our app pauses untrusted processes, checks whether speaker is mute and if so, re-posts the notification after turning it on to inform the user of the event (Section IV-A).

Within the Ward, our Guardian app continues to monitor any new process invoked in the background and any behavior changes that happen to existing processes. Whenever a process is found to become suspicious by the monitor and the detector (Section III-C), the controller immediately closes its app to protect the principal.

Exiting the Ward. When the operation of the principal is coming to an end, Guardian needs to wrap up the protection procedure, clean up the principal's computing environment whenever possible and resume the apps temporarily stopped. Most importantly here is to determine the timing for moving out of the Ward mode. A straightforward one seems to be the moment when the principal is switched to the background. This, however, could be caused by an event that is temporary but needs an immediate attention, such as an incoming phone call. To avoid mistakenly exiting the Ward mode, our implementation takes the following approach. In the case that the principal is replaced by a system or trusted third party apps in the foreground, and such an app is in communication category such as phone or Skype, the Guardian app does not rush to launch the exit procedure. It does this when the device is switched back to the *home* app and a pre-determined waiting period has expired. The period is set to avoid the user's accidental triggering of the exit procedure. On the other hand, whenever a third-party app starts to run in the foreground and the app is not in the communication category, our approach pops up a window asking the user's permission to exit the Ward mode and automatically does so after a short waiting period.

Once the exit procedure starts, Guardian first closes the principal, which removes the entire process directory of the app, making sure that its runtime statistics (such as its CPU, memory usages) there will not be exposed to unauthorized parties. Also based upon the user's setting, our app can clean all the caches using the permission `CLEAR_APP_CACHE`[3]. With such protection, however, still some app information cannot be easily cleaned up. Particularly, network-data usages sit under the `/proc/uid_stat/<uid>/` directory, which can only be reset when the device reboots. With proper protection during the protected app's runtime, what is left after its execution is just aggregated usage data after several rounds of communication, which are typically hard to use in an inference attack (Section IV). Guardian can further suggest to the user to deliberately take a few random actions (e.g., clicking on some random disease conditions in WebMD [22]) before exiting the Ward mode, making the chance of a successful inference even more remote.

After sanitization of the principal's runtime environment, Guardian runs its controller to launch the apps that have been closed. Specifically, Guardian uses `queryIntentActivities` to find out the main activity of the suspended apps, and then revives them through `startActivity`. Although this can be done to all apps, one can simply choose to only resume those that need to be recovered. As discussed before, the design of Android makes most background apps, those with 9 or above, disposable. Therefore, our approach only recovers the app with at least one process running at 5 or lower, while ignoring those with a high score. This treatment expedites the recovery process

[2]An app with a lower score is either system or foreground app.

[3]This is done using reflection to call the function `freeStorage`. Our current implementation needs to free all the caches, which affects other apps' performance. Therefore, the user is supposed to use it only when running the app with highly-sensitive information.

without affecting the utility of those apps (as those apps are supposed to run to their completion and can be terminated at any time), though it may come with some performance impacts, particularly when the user wants to run the recent program she used. In this case, the program needs to be restarted.

Utility Impacts. To understand the utility impacts of stopping third-party background processes, we analyzed 475 top apps from the Google Play store, in all 27 categories. Our analysis shows that a vast majority of the apps actually operate with high `oom_adj` scores, at level of 9 or above. This implies that once switched to the background, they can be terminated by Android anytime to make room for the foreground or system processes. Only 27 (5.68%) of those apps run at level 5 or below. Closing them may have some utility impacts, for example, stopping the background music. On the other hand, their runtime states are always well preserved when they are terminated. Specifically, for each of those apps, whenever we restart them, always they are restored to the states when they were killed, running at wherever they were stopped. Table IV (Section IV-B) presents a few examples for the analyzed apps. The findings indicate that the utility impacts of our approach are limited.

C. Finding Suspects

Even though most background processes are disposable and almost all of them can recover their states once restarted, this pause-resume approach still comes with some cost: invocation of an app takes a longer time than simply bringing a background process to the foreground, as illustrated in Table I; background apps could stop playing music or responding to Intents in the Ward mode. Limiting such performance and utility impacts relies on selection of right apps, those indeed suspicious, to close, avoiding blind killing of all apps. As discussed before, such selection is nontrivial, given that we cannot see detailed app behaviors such as system calls. In this section, we elaborate how we address this problem, using side channel information of individual apps to infer their activities.

App	Restart (s)	Switch (s)
Subway Surf	9.76	2.89
Mx Player	1.15	0.55
Flashlight	1.27	0.68
Shazam	2.18	0.77
RunKeeper	4.02	1.35
Bible.is	2.47	0.58
Chase	1.94	0.75
Duolingo	2.92	0.95
PicsArt	2.08	0.91
Wikipedia	1.91	0.65

TABLE I: Time of restarting an app vs. time of switching it to the foreground

Control strategy. As discussed before, our Guardian app is supported by a server that hosts information regarding how different apps should be protected. As examples, the phone app needs to be guarded against unauthorized audio recording; for WebMD, the concern is side-channel leaks. On the server side, such information is kept in a database, which includes types of threats a particular app is facing and policies for determining whether another app is suspicious of committing related attacks. In the above example, the policy for detecting an audio recording attack on the phone app can be the presence of an untrusted background app with audio record permissions, together with an `AudioIn_X` thread (for recording) created by `mediaserver`, as elaborated later in the section. This treatment leverages the features of a specific type of RIG attacks, which pose the major threat to the principal, to come up with a specialized defense strategy, and therefore avoids blindly killing all untrusted processes. On the other hand, when the user is looking for more generic protection or the information about the principal is not in the database, we can always roll back to the blind strategy that stops all untrusted apps. Here we describe how to defend against different RIG attacks (data stealing and side-channel leaks) in this way, based upon the apps' features, particularly their behaviors as inferred from their side channels.

Data stealing attacks. For some principals, the malicious app needs permissions to steal their sensitive data. Particularly, to record a phone conversation, the app must have the `RECORD_AUDIO` permission, and may also ask for `READ_PHONE_STATE` to get the incoming-call event (using the filter `android.intent.action.PHONE_STATE`). With these permissions, the app can tape the user's conversation by using the `MediaRecorder` object, which causes Mediaserver to create a service thread with the name `AudioIn_X`, where `X` is a number. Note that this name is generated by the OS and cannot be forged by the adversary. It discloses what the system is now doing (that is, recording audio on behalf of an app with proper permissions). The existence of such a thread can be directly observed from the public file `/proc/<pid>/task/<tid>/status`. Putting the pieces together, Guardian identifies a suspect for stealing phone conversations if the app has at least the `RECORD_AUDIO` permission and behaves in a suspicious way, that is, using CPU resources right before the emergence of an `AudioIn_X` thread. Here the CPU usage can be observed from `/proc/<pid>/stat`, which we discuss later. Our experimental study shows that this approach can timely stop the malicious app, preventing it from recording anything useful. Also, the strategy is very effective: when there is only one app with such a permission, we are almost certain that the app is malicious once the above condition is satisfied.

When it comes to official apps for critical Bluetooth devices, what we can do, again, is to closely monitor the background apps capable of stealing data. Here the apps are those with the Bluetooth permission. Specifically, once the principal (the official app) is invoked, Guardian periodically inspects all background processes to identify the ones with the permissions. In the meantime, our app also keeps track of the Bluetooth service process `com.android.bluetooth` by looking at its `/proc/<pid>/stat`. Whenever the process starts using CPU resources aggressively, Guardian immediately suspends all those untrusted, Bluetooth-capable apps (in the background) to protect the data of the principal running in the foreground. Note that even though the Bluetooth operations here could actually be caused by the official app itself, the observation nevertheless shows that its Bluetooth device is in the vicinity and some party has already started communicating with it. In this case, we have to stop untrusted Bluetooth apps in the background to protect the official app's data. Also interestingly, if the termination of the untrusted apps actually causes the Bluetooth service to stop, even temporarily, we have strong evidence that indeed at least one of these apps is malicious, trying to read from the device before the official app does, just like what is described in the prior research [6].

Side-channel attacks. The approach for detecting data-stealing attempts, however, does not work on side-channel attacks, which are more subtle and do not require any permission. A malicious app performing the attacks just collects public information from the principal's runtime. Finding such apps solely relies on analyzing their behaviors, which can only be inferred from the exactly same public resources (e.g., CPU usages) a malicious app utilizes to launch the attack.

A key observation of those side-channel attacks is that the malicious app has to continuously sample from its target's runtime environment (e.g., its CPU, memory, network data usages [1], its use of speaker [3] and the slight movements caused by touch-screen inputs to the target app [4]). Such sampling needs to be done fast enough to capture fleeting events, such as changes of memory footprints when a web page is being loaded, increments of `tcp_snd` when multiple packets are sent out [1]. Once the sampling rate goes down, the adversary starts to miss events that happen within the target and as a result, loses the granularity of observation necessary for inferring sensitive information. Therefore, a simple yet generic way to identify suspicious activities is just looking at how frequently a background app uses the CPU resources.

Apparently, we can get this information from the app's CPU usage (within `/proc/<pid>/stat`), which includes `utime` (the time spent on the user land in terms of clock ticks) and `stime` (the time spent on the kernel land). Here a clock tick is typically set to 10 milliseconds. The problem here is that the metric fails to describe how often an app is scheduled to use CPU. All we can do is to estimate whether the total usage here is sufficient for a RIG attack to succeed. This is hard because we have no idea how efficient the attack code could be. Further all CPU usages below one tick do not show up immediately. Therefore, we conclude that this information alone is not enough for identifying suspicious app behaviors.

What was used in our research is a new side channel, called schedule status (`/proc/<pid>/task/<tid>/sched stat`), which records the number of times an app has been scheduled to use CPU so far. This number provides precise information for determining the frequency the app uses CPU, which we call *Scheduling Rate* or *SR*. Specifically, an app's SR is the number of times it is scheduled to access CPU every second. As discussed above, to continuously monitor a target program, a malicious app must run at a certain SR level to achieve the necessary sampling rate[4]. Note that this does not mean that any app operating at this SR is necessarily monitoring the target. However, suspension of the background apps indeed scheduled too frequently helps protect the target (the principal) without blindly killing other processes, particularly when the information can only be collected at a high sampling rate. Further, as discussed before (Section III-B), closing an app typically does not affect its utility. Therefore, the cost for doing so, even to an actually legitimate app, is limited.

To further shorten the list of the processes that need to suspend, we just focus on the apps always active. For this purpose, our implementation of Guardian collects multiple (> 10) samples from each app's `schedstat`, one minute each, to calculate its average SR within that minute. When the principal is invoked, the app only closes the processes that have a significant number of samples (e.g., $> 30\%$) with SR above a certain threshold (once per three seconds in our experiment). In this way, we avoid suspending those only occasionally active in the background: for example, when the app receives a GCM message. Of course, these processes will continue to be monitored within the Ward mode, and be stopped whenever they increase their SR to a dangerous level. Also, for the apps with perceptible background activities like media players, Guardian first pauses such activities (e.g., music playing) and then stops the apps only if they are still active in one minute. Another piece of information leveraged by Guardian is kernel time `stime`. Most side-channel attacks need to continuously make system calls such as read from proc files, which raises the usage of the attack app's kernel time. Therefore, for the app that makes few calls, even when its SR is above the threshold, Guardian refrains from closing it as long as its `stime` goes below what is needed for a successful attack. A prominent example is the Amazon Shop app: one of its threads is scheduled at least twice per second; however, during its operation in the background (observed in one minute), we did not see any use of its kernel time.

Behavior change. Guardian is designed to identify the suspicious apps (e.g., based upon their SRs) and close them proactively, before entering the Ward mode. In response to this strategy, a malicious app may deliberately keep a low profile before the principal shows up in the foreground, and then act aggressively afterwards. The same approach was shown to be effective in keeping attack apps stealthy, according to the prior research [1], [20]. A distinct feature of this strategy is an observable correlation between the attack app's operation and that of the principal, which can be used to detect such a suspicious activity. Specifically, the Guardian app continues to monitor untrusted apps within the Ward mode, closing those that use CPU resources intensively and also comparing their behaviors with what have been seen outside the mode. An app is considered to be *stalking* the principal if its operations are found to be correlated to the principal's activities. This correlation is established through a statistical test on the activities of both the principal and the suspect, as elaborated below.

Specifically, we use Pearson correlation coefficient (r) to measure the correlation between two random variables X (the scheduling rate of the principal) and Y (the SR of the suspicious process). Several samples of X and Y are required to compute their correlation coefficient, which means that we need several instances of the suspicious app running side-by-side with the principal, with an elevated scheduling rate, while standing down once the principal stops. For realistic protection the number of samples should be very low (< 10). Actually, the number of samples for computing the coefficient depends on the value of the correlation coefficient r, power of the test $1 - \beta$ (β is the probability of type II error) and the significance level α (α is the probability of type I error). In the case that the correlation is strong (e.g., 0.9 or even close to 1), which is needed for the adversary to closely monitor the principal, the number of samples required for detecting such a correlation (at a given power and significant pair) can be very small (e.g., 4 times for the coefficient ≥ 0.98), as shown in Table II.

Once enough samples are observed and as a result, the correlation has been established, Guardian kills the suspicious

[4]In Android, a background process has a low priority, with little flexibility in adjusting its timeslice.

α	$1-\beta$	r	n
0.05	0.8	0.90	7
0.05	0.8	0.95	5
0.05	0.8	0.98	4
0.05	0.8	0.9993	3
0.05	0.8	1	3

TABLE II: Required number of samples for different values of correlation coefficient assuming 5% significance level (two sided) and 80% power of the test

app each time right before the system gets into the Ward mode, even when the app does not run aggressively (over the SR threshold). Also, our app alerts the presence of the suspect to the phone user. With her consent, the suspicious app can be uploaded to the server, which runs static and dynamic analyses on the program to find out whether it indeed aggressively accesses the principal's side channel information, for example, read from its proc files.

Collusion. If the adversary manages to get more than one malicious apps onto the victim's device, he might try to play a collusion game to make these apps look less suspicious. For example, in the phone tapping attack, a pair of apps, one with the `RECORD_AUDIO` permission and the other having `READ_PHONE_STATE`, can collude to make each of them less conspicuous. Of course, this trick does not work on Guardian, since it always terminates the app with `RECORD_AUDIO` whenever the `AudioIn_X` thread is observed. The situation becomes more complicated in the case of side-channel attacks, where two colluding apps could sample at a lower rate each but still collect sufficient information from the principal. The most effective way to mitigate this threat is to identify the relation between different apps. Such a relation can be captured by the referrals made between the apps, an approach the adversary needs to use to install more than one of his programs on the victim's device.

Specifically, Guardian automatically groups the apps signed with the same certificate and those whose installations are triggered by other apps: whenever an untrusted app invokes a marketplace app (e.g., Google Play, Amazon Appstore, SlideMe, etc.), which can be discovered by checking the app's activity stack, it is automatically linked to the new app installed from the marketplace. This approach was found to be very effective, never causing any false positive in our research. To circumvent the detection, the adversary could make a referral less obvious, for example, requiring the user to download the second app from a website. What we can do in this case is asking the user: as soon as an untrusted app is being installed, Guardian is notified through `action.PACKAGE_ADDED` and responds by popping up a view for the user to indicate whether the new app is recommended by an existing one; if so, an app list is provided for a convenient identification of the referrer. In this way, our approach can keep track of related apps installed *after* it starts running on the target device.

Also, Guardian operates a mechanism to detect colluding apps during their runtime. Actually, unrelated apps rarely use the CPU resources in an aggressive way together for an extended period of time. In our research, we selected 114 most popular free apps from Google Play and installed them on our Nexus 5. Among them, 68 were automatically invoked after the system rebooted. We further ran our monitoring app on the device to check these 68 apps' SRs every 5 minutes, for 40 hours. Only 15 app pairs, out of the total 1431 combinations, were observed to have combined SRs above 1 per 3 seconds (a threshold we set) over at least 10% of our monitoring period, while their individual SRs were below the threshold. A close look at these 15 pairs showed that the problem was actually caused by two apps `jp.naver.line.android` and `com.groupon` with a low privilege (7 to 9). Once they were suspended, none of the app pairs were found to have a collective SR above the threshold over 10% of the period. This indicates that we can temporarily suspend a selected app of an app pair once they are found to use the CPU resources together in an aggressive way for a while, without causing much utility trouble.

IV. Evaluation and Analysis

In this section, we report our analysis of App Guardian, in terms of its effectiveness in fending off known attacks, impacts on the utility of legitimate apps and the overheads it introduces during the operation.

A. *Effectiveness*

To understand the effectiveness of our App Guardian system, we evaluated our prototype against 12 RIG attacks, including those reported by prior research [1]–[5], [7], [20] and the new threats to Android-controlled IoT, as described in Section II-B. The study shows that our technique is capable of defeating all these attacks, at low cost most of time. Particularly, for the side-channel attacks, we show that using the scheduling rate to identify suspicious apps, Guardian effectively reduces the amount of information a RIG app can get from the principal. This result is illustrated in Figure 5. Also, the protection level can be balanced with the number of apps that need to be killed, which was also studied in our research (Section IV-B). Table III presents all the attacks evaluated or analyzed in our study. All the experiments were performed on a Google Nexus 5 (2.3G CPU, 2G memory). Following we report our findings.

No.	**RIG Attacks**	**Defeat**	**Attack Success Rate (SR)**
1	**Audio Recording**	Yes	N/A
2	**Bluetooth Data Stealing**	Yes	N/A
3	**Alarm Blocking**	Yes	Fail (2/s)
4	**Motion Detection On**	Yes	Fail (1/3s)
5	**WebMD**: inferring disease conditions	Yes	RG (1/2s)
6	**Twitter**: inferring identities	Yes	RG (end-to-end)
7	**Web Page Inference**	Yes	RG (10/s)
8	**Driving Route Inference**	Yes	Fail (1/s)
9	**Keylogger 1**: TouchLogger	Yes	$\leq$ 1/3s (1/3s)
10	**Keylogger 2**: Screenmilker	Yes	$\leq$ 1/3s (1/3s)
11	**Voice eavesdropping**	Yes[5]	Fail (1/3s)
12	**UI inference**	Yes[5]	Fail (1/3s)

TABLE III: Effectiveness in defending against RIG attacks. Here RG represents random guess. Keyloggers' success rate cannot go above 1 key per 3 seconds, given an SR of once per 3 seconds.

Audio recording and Bluetooth data stealing. We first put our system to the test against the data-stealing attacks. In the case of audio recording, we ran an attack app with both `RECORD_AUDIO` and `READ_PHONE_STATE` permissions. Whenever a phone call came in, the app started recording

[5]Our approach can defeat the attacks based upon the parameters given by their papers [2], [7].

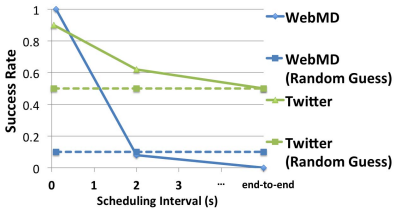
in the background. In the presence of App Guardian, however, this attempt was completely thwarted: as soon as the recording thread was spawned, the suspicious activity was immediately detected and the attack app was killed instantly. As a result, nothing was found to be recorded.

The Bluetooth attack was performed on iThermometer [28], a Bluetooth medical device also used in prior research [6]. Our attack app was successful in getting data from the device, right before its official app connected to it. However, this attempt no longer went through when the official app was protected by Guardian: as soon as the principal (the official app) was launched, Guardian detected the use of the Bluetooth service and the presence of a Bluetooth-capable app, and immediately terminated the app. This left the adversary no chance to collect any data from the the body thermometer.

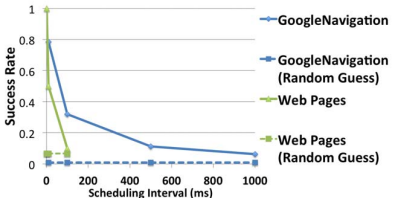
IoT attacks. We further used our Guardian app to protect the official apps of NetCam and Nest Protect (Section II-B) against the RIG attacks on them. For NetCam, we found that the chance for identifying the click on the "save clips" switch, which turns on or off its motion detector, decreases substantially with the attack app's SR: as shown in Table III, when its SR reduced to once every 1.5 seconds, the probability of correct detection goes down to 10%. This is because many packets (for unrelated purposes) actually all have similar sizes as the one for turning on the switch. The attack was successful because the packet is the only one that comes alone: no other packets show up in about one second since it arrives. When the attack process can only sample very slowly (1 per 3 seconds), this approach no longer works (which failed every time in our experiment) and actually the increment (`tcp_snd`) the adversary sees is very likely to involve multiple packets.

The "alarm blocking" attack (muting the sound for the alarm notification) is even more dependent on the attack app's SR: as soon as the GCM notification is discovered from its increments, the attacker is supposed to turn off the sound immediately, to prevent the recipient of the message from arousing the user's attention through an alert sound. We found that this can only be done when the app samples at least 20 times per second. What App Guardian does is to simply register with notification listener service, so that it is always informed whenever the official app posts a notification. When this happens, our app stops all untrusted apps that ran at SR of 20 times per second before the notification came and further checks the speaker status. If it is muted, Guardian unmutes it and further reposts the message, which produces the sound the adversary wants to avoid. Finally, the video watching part is hard to cover, due to the presence of a large volume of inbound traffic. To hide the information completely from the adversary, we can take the blind termination strategy, stopping all untrusted apps.

WebMD and Twitter. Similar to those IoT devices, WebMD and Twitter apps are also vulnerable to the RIG attacks that exploit their network-data usages [1] (also see Section II-A). Prior research shows that by monitoring detailed increments in their `tcp_snd` and `tcp_rcv` elements (at the packet level), the adversary can figure out the disease conditions the user checks through WebMD and the moment when the user tweets, which can be further used to recover her identity. In our research, we implemented the attacks described in the prior work [1] and then protected the targets of these attacks (WebMD, Twitter)



(a) WebMD and Twitter



(b) Google Navigation and Web Pages

Fig. 5: Effectiveness of RIG attacks under different scheduling rates. Here dash lines represent the success rate of a random guess.

with our App Guardian to understand the effectiveness of our technique. Here we report what we found.

Compared with the versions described in the prior research [1], both the WebMD and Twitter apps used in our study have been significantly modified. For WebMD, clicking a disease condition will generate a sequence of packets. Most of them change slightly (in 20 bytes) when the same condition is checked multiple times, while a few vary significantly in size, due to their inclusion of different advertising content. In our study, we randomly selected 10 disease conditions, and measured the range of the payload length for each packet associated with each condition. In this way, every selected condition here is fingerprinted with a sequence of payload-length ranges, as did in the prior work [1]. Using these fingerprints, we found that without protection, these conditions can still be uniquely identified from their payload-length sequences. However, once Guardian is in place, the side-channel attack becomes much more difficult to succeed. Specifically, when the SR of the attack app goes down to once every two seconds, all it observes is just an accumulated length of all packets related to one condition. In the presence of the randomness in packet sizes, the ranges of different conditions' accumulated lengths significantly overlap with each other. As a result, in all 25 random trials (each involving a click on one condition) we performed, only 2 led to correct identification of the conditions. Note that this identification rate is even lower than a random guess (1 out of 10 disease conditions). Also, when looking at the accumulated increments of `tcp_snd` and `tcp_rcv` "end-to-end", that is, from the invocation of the app until the moment that the user finishes checking one condition, we concluded that there is no chance to identify even a single condition (Figure 5a), as all the ranges of these accumulated increments (for different conditions) completely overlap with others.

The situation for Twitter is similar. The new version brings in randomness in packet lengths during sending and receiving tweets. Again, in our research, we fingerprinted those operations

using their individual sequences of payload-length ranges, and ran the side-channel attack [1] on the app. Even though the payload lengths associated with the same operation (send or receive) vary from time to time, we were able to correctly identify the tweeting activity most of time (45 out of 50) when the attack app runs at the SR of 20 times per second. However, with Guardian put in place, the SR was forced down to once every two seconds, which reduces the effectiveness of the attack to 62%, which is close to the random guess (determining whether it is sending or receiving tweets). We further studied the end-to-end situation, in which Guardian closes the attack app and restores it after the user sends or receives a tweet. From the accumulated increments, we found that these two operations cannot be separated, as the range of the increment in one case completely overlaps that of the other.

Web pages and driving routes. Prior research describes a technique to infer the web pages the user visits from the temporal changes in memory footprints when those pages are being rendered by the Chrome browser [5]. This attack requires the malicious app to sample the browser's memory dynamics (from the data resident size in `/proc/<pid>/statm`) at a high frequency. In our research, we followed what the researchers did: building up signatures for different web pages and then comparing the observed memory uses with the signatures to identify the pages visited. More specifically, such a signature is a sequence of tuples, each including a memory footprint size observed and its number of occurrences. The adversary's observation of memory uses is described as *memprint*, which is also a sequence of the tuples as the adversary sees. The attack happens by calculating a Jaccard index between a memprint (collected when the browser is loading a web page) and known signatures to identify the page.

In our experiment, we chose Alexa top 15 sites[6]. On our Nexus 5 phone, we first ran an attack program with the root privilege to collect signatures for these individual pages at a sampling rate of 10 per millisecond, which cannot be achieved by any ordinary background app. Then we reduced the sampling rate to find out whether the adversary can still differentiate these pages. For a rate of 100 per second, this was done easily. However, when the attack app ran at a realistic speed, 10 times per second in scheduling, which is what an ordinary app can possibly do, it only successfully identified the web pages at a probability of 10%, very close to a random guess (about 7%). This clearly indicates that our Guardian app can easily stop this attack without causing much collateral damage, since most background apps do not run that fast (Section IV-B).

Another side-channel attack reported by the prior research [1] is inference of driving routes from Android Navigator through the speaker's status (on or off). The idea is to continuously check the public API `isMusicActive` during navigation to identify the duration of individual voice elements for turn-by-turn voice guidance: e.g., "turn left onto the 8th street". The duration can be found because the speaker status turns from "off" to "on" at the beginning of the guidance and goes the other way around at the end. A sequence of these duration is a high-dimensional vector, which is used to search Google Maps for the path the user drove through.

[6]Note that the small set of pages we selected actually gives the adversary advantage: telling these pages apart does not mean that she is able to identify one website from millions of popular sites.

Most important to this attack is accurate measurement of such duration during the navigation app's runtime, which are compared with a set of reference sequences already mapped to certain routes. In our research, we collected 108 unique voice elements from 23 driving routes in a town and used their lengths measured at 200 per second as a reference. Then, we tried to match to them duration measured by an attack app implemented according to the prior work [1]. Accurate matches here are a *necessary* condition for a successful inference of these routes. In our study, we found that such an attack can be easily defeated by Guardian. Specifically, we measured the same set of the voice elements at different scheduling rates, which introduced errors to the measurement: i.e., the actual element length l now ranges from $l - \epsilon$ to $l + \epsilon$, when the sampling rate is $1/\epsilon$ caused by a given SR. With this error, we had to map such an element to any reference element within the $l \pm \epsilon$ range. As a result, the accuracy of the match decreases when the margin of the error grows, which happens when the attack app's SR goes down. In the experiment, we found that the attack went well at an SR of 100 per second: about 78% of the reference elements randomly measured were matched correctly. However, when the scheduling rate was dropped to once per second, only 6.3% elements were successfully matched. Note that under this accuracy level, it is impossible to match a sequence of elements to a right path on the map.

Keyloggers. We also analyzed the effectiveness of our Guardian app against keyloggers. Prior work shows that Android users' touch inputs can be revealed through a few attack techniques. Particularly, the smartphone's accelerometer discloses both shift and rotation data when the user types through touch screen, which is found to be informative enough for malware (e.g., Touchlogger [4]) to infer the key the user enters. Also, in the presence of a vulnerable screenshot app, Screenmilker can continuously capture the screen to determine the key being pressed [20]. Despite the diversity of the techniques used in such attacks, the chance for these keyloggers to successfully identify the user's inputs depends on their sampling rate. Consider an Android user's average typing speed of 3 keys per second. When the sampling rate goes down to once per second, the best the adversary can do is just to pick up 1 of these 3 keys. Note that this success rate is all but unattainable for the adversary, as the malware typically needs more than one sample to correctly figure out one key. For example, Touchlogger uses multiple device orientation data to extract features of one keystroke; Screenmilker, on the other hand, needs to continuously take shots in order to catch the moment when a key is entered. In those attacks, the malicious app has to be scheduled at least once whenever it takes a sample (shift and rotation data or screenshot). Therefore, with the SR decreases (e.g., to 1 per 3 seconds), the amount of information those Keylogger can obtain is very limited (no more than one key every 3 seconds). Obviously, they get nothing in the case of end-to-end protection.

Voice eavesdropping and UI state inference. Recently, a technique has been proposed to utilize the smartphone's onboard gyroscope to eavesdrop on the user's phone conversation [7]. The gyroscope is sensitive to audible signals that range from 20Hz to 200Hz. To catch such signals, the attack app needs to collect gyroscope readings at a very high speed: specifically, we ran the attack code made public by the authors of the paper [29], and found that its SR is 20 times per second, much higher than the threshold utilized by Guardian (1 per 3 seconds).

In the presence of the phone app, the attack app running at such speed will be suspended by our app. On the other hand, once we force the malware to get only one reading every three seconds, clearly little can be inferred about the ongoing phone conversation.

Another recent RIG attack is to infer the state of the target app's user interface (UI) for a phishing attack, using the app's shared memory information collected from its proc file [2]. In the paper, the authors indicate that the SR of their attack app was between 10 to 33 times per second for identifying the target app's activity transition. More importantly, the malware needs to accurately determine when an activity is about to launch and then inject into the foreground a phishing activity to steal the user's sensitive information, e.g., the password she is supposed to enter into a login activity. Under the protection of Guardian, the malware has to reduce its SR below 1 per 3 seconds to avoid being terminated. At this sampling rate, it is conceivable that the malware cannot observe the transition between two activities, which completes in sub-seconds, not to mention identifying the right moment for hijacking the target's login activity.

B. Utility Impacts and Performance

We further studied the utility impacts of App Guardian on legitimate apps and its performance. For this purpose, we used 475 apps from 27 categories on the Google Play store. Examples of these apps are described in Table IV. As discussed before (Section III-B), all these apps are top-ranking ones in their individual categories, including Facebook in Social, Pandora in Music & Audio and Amazon in Shopping. Among all these apps, 27 apps get `oom_adj` values of 5 or lower once switched to the background. Most of them are media players and keyboard apps, and the rest are launchers and weather apps (Section III-B). All other apps are assigned a higher value (usually 9 or above), which indicates that they can be killed at anytime without affecting the system's normal operation and user experience [27]. Further, all these apps are capable of restoring their states after being terminated. Therefore, killing their processes just temporarily stops their services, which can be resumed later on.

App	Category	SR	oom_adj	Recoverable
Facebook	Social	$< 1/3$	9	Yes
Fox News	News & Magazines	$< 1/3$	9	Yes
Yelp	Travel & Local	$< 1/3$	9	Yes
Viber	Communication	$1/1$	5	Yes
Amazon	Shopping	$2/1$	9	Yes
The Weather Channel	Weather	$< 1/3$	9	Yes
FIFA	Sports	$< 1/3$	9	Yes
Temple Run 2	Games	$10/1$	9	Yes
Photo Grid	Photography	$< 1/3$	9	Yes
Adobe Reader	Productivity	$< 1/3$	9	Yes

TABLE IV: Analysis of top ranking apps (examples): here SR is described as number of schedules per seconds.

Impacts on popular apps. In our experiment, we measured the scheduling rates of such popular background apps. Two minutes after they were switched to the background, we monitored their SRs for 5 times. We found that totally 183 (38.5%) out of 475 apps were scheduled at a rate over once every three seconds for at least once and 135 (28.4%) apps for all five times. However, this high SR (1 per 3 seconds) did not last for every app: 43 out of the 135 apps no longer utilized CPU at this rate after 30 minutes. Among the rest 92 persistent fast-running apps (19.3%), 77 were assigned an oom_adj of 9 or above, essentially being marked as disposable by Android; 7 were assigned a score between 6 and 8, and therefore can be terminated without causing serious utility impacts; the rest 8 were executed with a value of 5 or below, including 6 media players, 1 video chat app and 1 launcher app. For the media players, Guardian had to pause their music (Section III-B) while running the principal in the foreground. As a result, we found that 4 of them no longer used CPU aggressively and therefore did not need to be killed. The rest 4 were terminated but later restored after the principal completed its operation. Also, all apps also ran smoothly after the recovery.

Overall, among all the popular apps, Guardian only needs to suspend 19.3% of the apps. The vast majority of such suspensions have little observable impact on the utility of these legitimate apps at all, as they are considered to be disposable, even though re-invoking these apps will take a longer time. Only a very small portion, about 1.68% (8 out of 475), once stopped, may slightly affect the phone users' experience (music temporarily stopped, status of online chat app temporarily goes offline, etc.) but they can all be recovered once the system moves out of the Ward mode.

Overhead. We further measured the performance of our Guardian app on 2 Nexus 5 phones, each installed with more than 250 apps. Using OS Monitor [30], our prototype was found to work efficiently. It took only 5% of the CPU resources and 40 MB of memory under the Ward mode, and the CPU usage dropped to as low as 1% after leaving the Ward mode, as observed in our experiments. The battery consumption of Guardian is also low, which was measured in our study on 2 Nexus 5 phones with 50 top apps running on each of them. From the battery statistics provided by Android, we found that Guardian consumed 0.12%, 0.18% of the total battery capacity per hour within the Ward mode and 0.75%, 1.05% per 24 hours otherwise. For example, consider that Guardian enters into the Ward mode 12 times a day and 5 minutes each, it will use about 0.84% and 1.18% of the battery on these two devices. This is lower than running the Facebook app for 30 minutes a day (about 1.2% of the total battery consumption).

V. Discussion

Detection and separation. App Guardian is *not* a malware detection system. All it finds is just suspicious programs that meet a set of necessary yet often insufficient conditions for a RIG attack. The idea is to suspend a small group of apps to minimize performance and utility impacts on the system's normal operation and user experience. Even if we get it wrong, terminating legitimate apps, they can still be restored to the original states after the principal's execution. Saying that, this approach does bring in a certain level of inconvenience to the user, who could experience a delay when switching to the app she just runs or the stop of background services when using her protected app. Therefore, a more accurate identification of malicious activities, which helps further narrow down the list of apps that need to kill, certainly helps. To move forward, we expect a further investigation on the real-world impacts of the whitelist and the behavior-based app selection strategy on real users' devices, to understand indeed how much inconvenience the user will perceive when using our new technique. Also, we

will look into other side channels (e.g., process states) to gain as much insight as possible into an app's operation.

Another concern is the potential for the adversary to evade our protection, for example, through adding perceptible activities into the attack app to prevent it from being killed. Our preliminary study shows that for the common apps with such features, like media players and keyboards, Guardian can first stop their perceptible activities and then terminate them. Further studies are expected to better understand what tricks a malicious app can still play to bypass our protection mechanism.

Background process protection. The current design of App Guardian is for protecting security-critical foreground apps. Such apps only run within a short period of time and can therefore be secured by pausing suspicious background apps. Although most apps that need protected indeed run in the foreground, there are situations where a background process is also under a RIG threat: an example is when the GCM process delivers a notification to apps and the observation of the notification itself already leaks out information. Background services may run indefinitely, so they cannot be protected in the same way as the foreground process. Further effort needs to be made to understand whether protection of such a process can be done at the app level and if so, how to do it at the minimal utility and performance cost.

Sanitization. Another issue that needs a further investigation is whether it is possible to thoroughly clean up the principal's execution environment after the program stops running. As discussed in Section III-B, information such as accumulated network-data usages of an app cannot be removed without rebooting the whole device. Adding noise to the data also needs the user's intervention. A question is how to better protect such data and whether this is feasible without touching the OS and the app under protection. This should be studied in the follow-up research.

VI. Related Work

Data stealing attacks and defense. With more and more private user data moving onto mobile devices, they increasingly become the main target for data-stealing attacks. These attacks often exploit the design limitations of Android, which does not provide fine-grained access control. For example, an app given the `RECORD_AUDIO` permission can make a record at anytime, even when a sensitive phone call is ongoing. As another example, any app with the `BLUETOOTH` permission is free to access any Android Bluetooth accessories, including medical devices [6]. Other attacks also in this category include information leaks due to the weakened memory randomization protection on Android [31] or insufficient protection of content providers [32]. Mitigating such a threat usually relies on modification of the operating system. For example, prior research [6] shows that the Android Bluetooth service can be hooked to prevent the attempt to gain unauthorized access to medical devices.

Side channel attacks and defense. Side channel attacks has been studied for decades and new channels are continuously discovered [33]–[36]. Most of the time, those attacks are also RIG, as the attack process needs to continuously collect information from the target program during its runtime. Particularly, prior research [37] shows that sensitive user information can be collected from the Linux proc file systems: through sampling the target program's `ESP/EIP` changes, inter-keystroke timings can be identified to infer the user's inputs. Memory usage is also found to leak sensitive information: Momento [5] utilizes `/proc/[pid]/statm` to find out the websites visited by the victim. The paper briefly mentions an approach that infers the user's inter-keystroke timings using schedule status. However, the attack was only performed on a desktop, since the authors seem to believe that this piece of information was not available on Android [5]. Actually, it has been there since 2.3 (or even earlier). Most importantly, we are the first to leverage this side channel to infer mobile apps' behaviors for the purpose of *defending against side-channel attacks*. Also related to our work are the study on `shared_vm` and `shared_pm` (for inferring the UI state of an Android app [2]) and the research on network-data usage, audio usages, etc. (for identifying one's identity, disease, locations and finance) [1]. In addition, sensors on smartphones have been exploited to collect sensitive user information [17], [19]. Examples include Soundcomber [3] that uses audio to find credit-card information, Accomplice [18] that leverages accelerometer for location identification and Touchlogger [4] that also utilizes accelerometer for key logging.

So far, almost all existing defense techniques against side-channel attacks require change of either operating systems or vulnerable applications [1], [5]. Up to our knowledge, App Guardian is the first third-party app level protection that has ever been proposed. Notably, HomeAlone [38] is the only work we are aware of that uses side-channels for defensive purposes. It verifies a virtual-machine instance's exclusive use of a physical machine through the cache channel. By comparison, our approach is designed to protect mobile systems against side channel attacks. For this purpose, it leverages a set of unique side channels that have never served this purpose.

VII. Conclusion

In this paper, we report our study on an emerging security threat to Android, the runtime-information-gathering attacks, which cover a wide spectrum of new attacks that aim at exploiting apps for sensitive user data, ranging from phone conversations to health information. Our research provides further evidence for the seriousness of such a RIG threat, showing that popular Android-based IoT systems are equally vulnerable to this type of attacks. Mitigating this emerging threat needs to thwart a malicious app's attempt to run side-by-side with the principal, in an attempt to collect its runtime information. This is achieved in our research without changing the operating system and the principal. Instead, we use an ordinary app, Guardian, which pauses suspicious background processes when the principal is running and resumes them after the security-critical operation is done and the environment is cleaned. We show that this approach does not damage the utility of legitimate apps due to the observation that most background apps on Android can be stopped without disrupting their functionality. To further reduce the inconvenience of doing so, Guardian utilizes a set of novel side channels to infer background apps' behaviors and identify a small set of them that meet necessary conditions for the RIG attacks. In this way, most third-party apps can still run without being interrupted. Our evaluation shows that our approach works effectively against all known attacks, at a minimal performance and utility cost.

We believe that our technique significantly raises the bar for the RIG attacks, a realistic threat to mobile security, and the research on this subject. The idea of side-channel based detection and the lightweight response for mitigating negative impacts of a false alarm can further inspire the follow-up effort on developing app-level protection against other security threats on mobile devices.

ACKNOWLEDGEMENTS

We thank our shepherd Matthew Smith and anonymous reviewers for their comments and help in preparing the final version of the paper. The project was supported in part by the NSF CNS-1117106, 1223477 and 1223495.

REFERENCES

[1] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, location, disease and more: Inferring your secrets from android public resources," in *Proceedings of 20th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2013. [Online]. Available: http://www.cs.indiana.edu/~zhou/files/fp045-zhou.pdf

[2] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it: Ui state inference and novel android attacks," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 1037–1052. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/chen

[3] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones." in *NDSS*. The Internet Society, 2011. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss2011.html#SchlegelZZIKW11

[4] L. Cai and H. Chen, "Touchlogger: inferring keystrokes on touch screen from smartphone motion," in *Proceedings of the 6th USENIX conference on Hot topics in security*, ser. HotSec'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028040.2028049

[5] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 143–157. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.19

[6] M. Naveed, X. Zhou, S. Demetriou, X. Wang, and C. A. Gunter, "Inside job: Understanding and mitigating the threat of external device mis-bonding on android," 2014.

[7] Y. Michalevsky, D. Boneh, and G. Nakibly, "Gyrophone: Recognizing speech from gyroscope signals," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 1053–1067. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/michalevsky

[8] "Smartthings," http://www.smartthings.com/, 2014.

[9] "Nest," http://www.nest.com/, 2014.

[10] "Automatic, an auto accessory to mak you smarter driver," https://www.automatic.com/, 2014.

[11] "Viper smart key," http://www.viper.com/, 2014.

[12] "Doorbot," http://www.getdoorbot.com/, 2014.

[13] "Netcam," https://netcam.belkin.com/, 2014.

[14] "Nest saw 'tens of thousands' of its smart smoke alarms come online within two weeks," http://www.forbes.com/sites/parmyolson/2013/12/05/nest-saw-tens-of-thousands-of-its-smart-smoke-alarms-come-online-within-two-weeks/, 2014.

[15] "Demo: Leave me alone: App-level protection against runtime information gathering on android," https://sites.google.com/site/appguaridan/, 2014.

[16] "My data manager," http://www.mobidia.com/products/takecontrol/, 2014.

[17] Z. Xu, K. Bai, and S. Zhu, "Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WISEC '12. New York, NY, USA: ACM, 2012, pp. 113–124. [Online]. Available: http://doi.acm.org/10.1145/2185448.2185465

[18] J. Han, E. Owusu, T.-L. Nguyen, A. Perrig, and J. Zhang, "Accomplice: Location inference using accelerometers on smartphones," in *Proceedings of the 4th International Conference on Communication Systems and Networks*, Bangalore, India, 2012.

[19] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury, "Tapprints: Your finger taps have fingerprints," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 323–336. [Online]. Available: http://doi.acm.org/10.1145/2307636.2307666

[20] C.-C. Lin, H. Li, X. Zhou, and X. Wang, "Screenmilker: How to milk your android screen for secrets," 2014.

[21] "Android permission," http://developer.android.com/guide/topics/manifest/permission-element.html/, 2014.

[22] "Google play: Webmd for android," http://www.webmd.com/webmdapp, 2012.

[23] "Yahoo finance," https://play.google.com/store/apps/details?id=com.yahoo.mobile.client.android.finance&hl=en/, 2014.

[24] "Best 20 internet of things devices," http://techpp.com/2013/10/16/best-internet-of-things-devices/, 2014.

[25] "Android oom killer," http://elinux.org/Android_Notes#oom_killer_info, 2014.

[26] "getmemoryclass," http://developer.android.com/reference/android/app/ActivityManager.html#getMemoryClass(), 2014.

[27] "Processlist.java," https://android.googlesource.com/platform/frameworks/base/+/master/services/java/com/android/server/am/ProcessList.java, 2014.

[28] "ithermometer," http://www.ithermometer.info/, 2014.

[29] "Gyrophone code," https://bitbucket.org/ymcrcat/gyrophone/, 2014.

[30] "Os monitor," https://play.google.com/store/apps/details?id=com.eolwral.osmonitor&hl=en/, 2014.

[31] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee, "From zygote to morula: Fortifying weakened aslr on android," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 424–439. [Online]. Available: http://dx.doi.org/10.1109/SP.2014.34

[32] Y. Zhou and X. Jiang, "Detecting passive content leaks and pollution in android applications." in *NDSS*. The Internet Society, 2013. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss2013.html#ZhouJ13

[33] D. X. Song, D. Wagner, and X. Tian, "Timing analysis of keystrokes and timing attacks on ssh," in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, ser. SSYM'01. Berkeley, CA, USA: USENIX Association, 2001, pp. 25–25. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267612.1267637

[34] J. Trostle, "Timing attacks against trusted path," in *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, May 1998, pp. 125–134.

[35] L. Zhuang, F. Zhou, and J. D. Tygar, "Keyboard acoustic emanations revisited," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 3:1–3:26, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1609956.1609959

[36] M. Vuagnoux and S. Pasini, "Compromising electromagnetic emanations of wired and wireless keyboards," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855769

[37] K. Zhang and X. Wang, "Peeping tom in the neighborhood: keystroke eavesdropping on multi-user systems," in *Proceedings of the 18th conference on USENIX security symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 17–32. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855770

[38] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "Homealone: Co-residency detection in the cloud via side-channel analysis," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 313–328. [Online]. Available: http://dx.doi.org/10.1109/SP.2011.31