Journal of Computing and Communication Vol.1 , No.2 , PP. 24-43 , 2022

# A New Security Defense Approach for Android Via Proactive Restart

Zhiyong Shan*[a], Iulian Neamtiu[b]

[a]School of Computing, Wichita State University, Kansas, USA, [b]Department of Computer Science, New Jersey Institute of Technology, New Jersey, USA

*Corresponding Author: Zhiyong Shan [*Zhiyong.shan@wichita.edu*]

ARTICLE DATA

***Article history:***
***Received 24 Mar 2022***
***Revised 08 August 2022***
***Accepted 09 August 2022***
***Available online***

*Keywords*:
*Security Defense*
*Android*
*Proactive Restart*

ABSTRACT

The pervasive use of smartphones requires novel approaches to defend against many zero-day smartphone attacks. In this work, we propose a novel proactive approach to stop certain categories of attacks on smartphone apps. The key insight of our approach is to exploit the asymmetry between the high-level state where user-app interaction takes place and the low-level state that attackers target. Specifically, we leverage a smartphone's native support for quick and lossless restarts -- an action that is minimally intrusive for users but disruptive and confusing for attackers. We show how our approach thwarts two classes of attacks -- Activity Inference and Task Hijacking. Experiments on 34 popular Android apps using three proactive restart strategies have revealed that our approach is effective at reducing side-channel time series predictability (hence increasing attacker's burden) and efficient at imposing an acceptable overhead. Restarts also can change back stack contents and thus can help detect Task Hijacking attacks. We propose a time-series entropy metric to quantify resilience to known and unknown attacks. Our experiments show that our tool can detect four types of Task Hijacking attacks.

## 1. Introduction

Cyber-security defense can be either reactive (wait for the attack to be detected, then take measures to stop, contain, and mitigate the attack) or proactive (continuously change the attack surface to prevent attacks in the first place — this strategy is also known as cyber maneuver (CM) [1]). Given the diversity of modern attacks and the lack of effective detection mechanisms [2], reactive approaches are becoming less viable as defense strategies. In contrast, proactive approaches such as cyber maneuver, in which the defender continually changes the attack surface to deflect potential attacks and make the attacker's job harder, are particularly suited for defending against known and unknown (zero-day) attacks. However, CM comes at a cost, e.g., in terms of time and resources, because the system is maneuvering even in the absence of attacks. Hence achieving effective CM involves balancing the cost of maneuvering with the benefit of reducing attack risk [3].

Smartphones are increasingly used in sensitive or critical fields, such as military, banking, e-commerce, or government. With increased smartphone use, the potential for smartphones to be the target of attacks also increases. For smartphones used in environments with infrequent updates (e.g., due to company policy, user preferences, or the lack of a highly available, trusted channel for downloading and applying security patches), the problem is further compounded, suggesting that a live, on-the-phone, proactive security approach is needed.

In this paper, we propose a novel CM mechanism that makes headway toward achieving proactive security for smartphones via a simple yet effective proactive restart approach. Our intuition is to exploit the asymmetry between user-perceived app-state (high level) and attacker-observable state (low-level). Our key insight is that on smartphones, the high-level application state is naturally preserved across application restarts. In contrast, the application's low-level operating system (OS) state is cleaned up, destroyed, or otherwise substantially perturbed. However, most attacks target the OS state, as it is rich in side channels. Therefore, our app restarts are natural for the application but

disruptive for the attackers; hence proactive restarts de facto implement CM, protecting the application without significantly degrading its functionality.

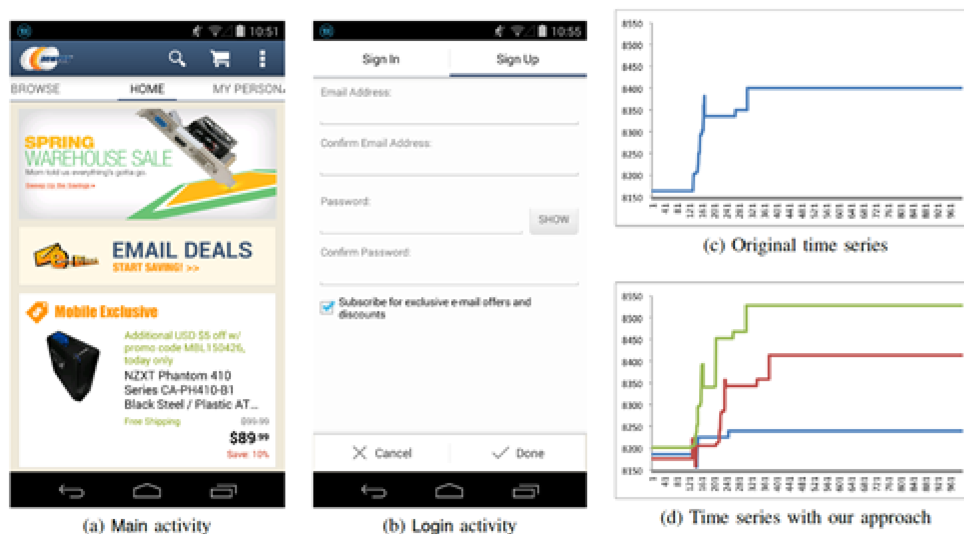



FIGURE 1. Source activity (left) and destination activity (center); /proc/[pid]/statm shared memory size (KB) time series (right) .

On smartphone platforms such as Android and iOS, applications ("apps") are subject to frequent pause/resume (or stop/restart) operations. For example, whenever the user switches to another app, turn the screen off, changes screen orientation, or the phone is running low on memory, the smartphone OS will pause or stop the current app (unless the app is providing a background service); conversely, when the user turns the screen on, or switches back to the original app, the app is resumed or restarted. Hence these pause/resume or stop/restart operations are a core, "first class" functionality of smartphone OSes. To provide this first-class support, smartphone OSes automate many state management tasks, e.g., quickly and automatically saving and restoring graphical user interface (GUI) state across restarts and invoking user-defined callbacks for saving and restoring app state. Thanks to platform support, restarts efficiently provide a smooth user experience. This is in stark contrast to desktop or server programs, where programs are effectively "running forever"; hence state save/state restore operations are ad-hoc, and OSes offer little support. Given smartphones' first-class support for quick and lossless restart, our key insight is to use restart as a CM strategy with modest costs.

The Android platform and apps have been subjected to various security attacks [2], and various defenses have been proposed. Many classes of attacks rely on observing the victim app for some time, inferring its behavior, or learning operational parameters that facilitate launching the attack. For example, in an Activity Inference attack [4], a malicious app M running in the background can infer the current screen of a victim app B by monitoring B's runtime parameters, as exposed in the /proc filesystem. However, if B's parameters are unavailable, obfuscated, or random, whole attack classes will be ineffective. Note the high-level vs. low-level asymmetry: since attackers cannot directly observe high-level (e.g., GUI) state, they target low-level (e.g., OS) state instead and try to infer the former.

Drawing on these insights, we propose a new approach for protecting Android apps via proactive restarts: we proactively trigger pause/resume operations to confuse and protect against attacks. As mentioned previously, on Android, pause/resume or stop/restart operations are quick and non-intrusive, given the native platform support for pause/resume and stop/restart. Consequently, proactive restarts are fast and unobtrusive to the user but disruptive at the OS level. As discussed in Section II-A, this disruption spans process resources from identifiers to files, memory, and IPC. This disruption confuses the attacker as the process parameters change; now, the attacker has a harder time inferring application behavior, and the partial information the attacker has gathered about a running process is stale or even useless, e.g., when the old app process is killed, and a new process is started.

In Section II we introduce background knowledge of the paper, including the restart/resume mechanism of Android, Activity Inference Attacks, and Task Hijacking Attacks. Section III provides concrete examples of how the

attacks mentioned above (Activity Inference and Task Hijacking) can subvert the popular Newegg app and how our approach hinders such attacks. In Section IV, we discuss our Implementation. We use Android's native restart capabilities and app management services to implement our proactive restart scheme. Section V evaluates our approach on 34 Android apps chosen from various categories.

Table1:
Android restart levels.

| Level | Cause | Changes |
| --- | --- | --- |
| 1: Pause activity | The activity becomes (partially) covered; Turn off the screen | User space: activity status information in Android Activity Manager Service (AMS) and Windows Manager Service (WMS) is changed. |
| 2: Stop activity | Switch to another app; Start a new activity in the same app; Receive a phone call; Press the' Home' button | User space: activity status information in AMS and WMS is changed; activity stack in AMS is changed.<br>Kernel space: changes to process resources, e.g., shared memory, files, sockets, and semaphores. |
| 3: Destroy activity | Press the 'Back' button; Kill the app | User space: activity is removed, and new activity is created in AMS and WMS; the activity stack in AMS is changed.<br>Kernel space: the process is killed, and a new process is created in the OS. The completely new process, e.g., new process identifier, memory mapping, and resources. |

In summary, our main contributions are: 1) A novel, proactive approach for cyber maneuver based on the insight that smartphone app restarts are frequent and lossless but perturb OS state. 2) A formulation of attack resilience using time series entropy. 3) An evaluation of the proposed approach on 34 Android apps.

## 2. Background

We now present background information on proactive security and the resume/restart mechanism on Android.

### 2.1. Android Restart

The Android smartphone platform consists of apps, usually written in Java, running on top of the Dalvik virtual machine (Android versions < 5.0) or the Android runtime (ART versions 5.0); these, in turn, run on top of a smartphone-specific Linux kernel. Due to the platform's nature, Android apps are centered around a GUI; an app's GUI consists of separate "Activities", where an activity roughly corresponds to a screen in a desktop program's GUI. As a result of user interaction or outside events, an app transitions among activities; for example, in the Newegg online shopping app, if the user is in the Main activity and clicks the 'My Account' menu item, the app transitions to the Login activity (see Figure 1).

Smartphones (unlike desktop or server systems) have limited resources. When the system is low on memory, or the user turns the screen off or switches to a different app, the current app is automatically paused or even killed; a small percentage of apps that provide background services remain running, albeit in a restricted mode. The app is resumed or restarted when the user returns to the app. Hence, smartphone apps and OSes are designed from the ground up to support pause/resume operations smoothly and efficiently.

In Android, our target platform, there are three main restart levels. We present these levels in Table I, which consists of three columns: levels, causes, and changes. A restart cycle has an impact on both the app and operating system kernel. Especially a restart cycle at level 3 — destroy app, is very disruptive, as the process is killed. When

the app restarts, it restarts with a new process and new activity in OS kernel, AMS, and WMS. Since restart is such a common and efficient operation on smartphones and is gracefully tolerated by apps while being disruptive for the OS, our key insight is to use proactive restarts to change the attack surface hence offering a cyber maneuver capability.

### 2.2. Activity Inference Attack

Activity inference [4] represents a class of side-channel attacks where a malicious background application M can stealthily infer an activity transition occurring in a foreground benign app B. Further, M can precisely pinpoint which activity B is transitioning into in real-time. The attack is strong as it does not require any special permission. There is no vulnerability being exploited since all the information gathered by the malware M is publicly available, including /proc files, e.g., /proc/[pid]/statm.

The fundamental weakness of such attacks is that the information exposed through such channels correlates well with B's activity transition behaviors. For instance, when an activity transition occurs in the foreground, the application process allocates a screen buffer for the new activity as shared memory with a fixed size (proportional to the screen size) and then deallocates the buffer of the previous activity. Such unique memory consumption patterns can be easily captured through the /proc side channel. Furthermore, each destination activity has a different initial behavior, e.g., some activity's onCreate() callback may load an advertisement and therefore cause a new network connection to be created. Through other side channels, such initial behaviors distinguish the destination activity.

The Activity Inference attack has many consequences. Once the background malware M infers which foreground activity B is transitioning into, it can inject a phishing activity into the foreground to preempt B. The user will then be fooled into interacting with the malware M instead of the original app B.

Our scheme aims to address this fundamental weakness by using proactive restart to produce changes in OS state that are harder to predict, undermining the attacker's assumption that the side channel is reliable.

### 2.3. Task Hijacking Attacks

*The back stack:* In Android, each app (or "task") consists of an activity stack (aka "back stack") where activities are ordered by access time. While there are multiple tasks on the phone, only one is displayed on the screen at a time – the foreground task. The activity on the top of the back stack of the foreground task is called foreground activity. If the user clicks the 'Back' button, the foreground activity is destroyed, and the activity beneath it will be displayed.

*Task hijacking attacks:* Certain flags can be used to deviate from the strictly stack-based activity transition order. For example, task affinity manipulates the back stack order while allowTaskReparenting relocates an activity to another task's back stack. Task hijacking attacks [5] represent a class of attacks launched by abusing these task state transition conditions. Attackers may steal login credentials, implement ransomware, and spy on users' activities.

Concretely, the malicious task stays in the background while the victim task is in the foreground. When the user makes an activity transition, an activity from the malicious task will be put in the first or second position in the stack of the foreground task. Note that the user is not aware of the abnormal change of the task stack and thus considers the malicious activity benign. Then the malicious activity can further achieve some forms of attacks that include spoofing, phishing, ransomware, or preventing app uninstall.

*Our approach:* Our scheme detects this class of attacks by using a proactive restart to produce changes in the back stack of the foreground task. After restart, the compromised foreground task will exhibit a particular abnormal behavior. Then we can determine that an attacker has maliciously changed the task's back stack.

## 3. Sample Attacks

We now present two examples that motivate our approach and illustrate our solution.

### 3.1. Activity Inference Attack

Consider the Newegg Mobile app. An attacker might use an Activity Inference attack to determine which activity Newegg Mobile is in and which activity it is transitioning to. The attacker can inject its own fake activity to try phish secrets.

Let us suppose that Newegg Mobile is in the Main activity (Figure 1 (a)) and is preparing to transition to the Login activity (Figure 1 (b)). An Activity Inference attack relies on observing side-channel information, i.e., shared memory

values in /proc/pid/statm; for an unprotected app, the time series of shared memory is presented in Figure 1 (c). The transition event is clearly distinguishable in the time series, as it is a single event. If the attacker detects this event quickly, then the attacker can "pop up" a fake activity that looks very similar to log in, and trick the user into inputting data into the fake activity — if this input data is sensitive information, such as a username/password combination (as is the case here), a credit card number, or a bank account number, the attack succeeds.

However, since our approach injects restart events, the time series of shared memory values, shown in Figure 1 (d) is confusing for the attacker: due to the perturbation introduced by restart, depending on where we choose to restart, there can be multiple time series with multiple events (Figure 1d, red and green curves, which represent strategies S3 and S4 defined in Section IV). Our approach can deliberately insert restart events into the current activity just to confuse the attacker into believing there is an activity transition occurring when in fact there is no such transition (Figure 1d, black curve, which represents strategy S2 from Section IV).

Hence our proactive approach confuses the attacker into not knowing if, and when, the app is transitioning between activities.

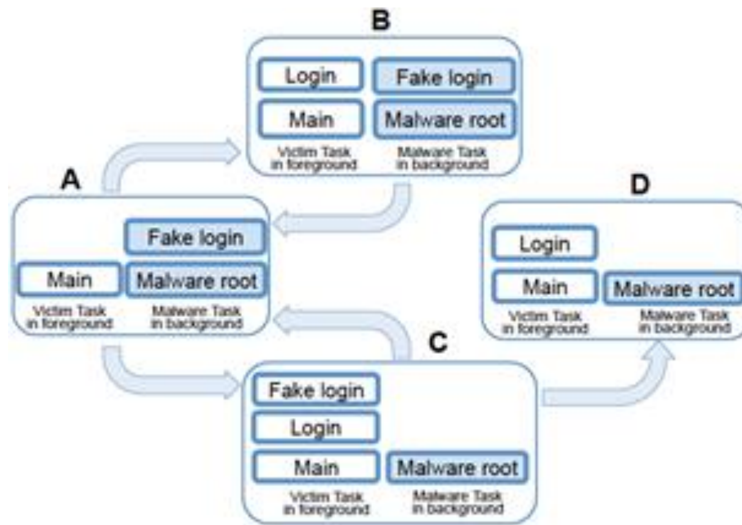



Figure 2: Task Hijacking attack: task state transitions.

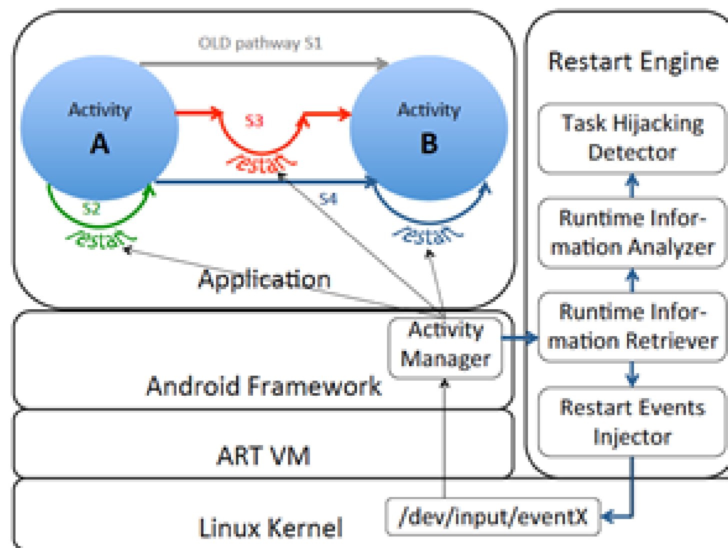



Figure 3: Overview of our Implementation.

## 3.2. Task Hijacking Attack

Suppose that instead of the Activity Inference attack, the attacker launches a Task Hijacking attack in the same state, that is, at the A to B transition. The end goal of the attack is to place the fake Login activity in the foreground. Figure 2 shows the task state transitions of the attack. State A is before activity transition, corresponding to Figure 1 (a). The Main activity is shown on the screen, but two malware activities are in the background. State B is the result of a normal activity transition. The new Login activity is created and shown on the screen. If the user presses the 'Back' button, the Login activity is destroyed, and the state goes back to A. State C results from a Task Hijacking attack. The new Login activity is created while the fake Login activity is relocated to the foreground task and covers the real Login activity. But the user is not aware of the fake Login. Thus the fake Login can steal the account name and password. Suppose the user presses the 'Back' button, and the state switches to D. At this point. In that case, our hijacking detector prevents the transition (Section IV-B) – an illegal activity transition, due to an attempted attack.

Our solution is to proactively restart the fake Login activity and monitor for changes in the back stack. When restarting at destroying level, the state transitions from C to D, displaying the activity beneath the fake Login. If the app transitions to A, our approach detects this as abnormal behavior. Hence, our proactive approach detects Task Hijacking attacks.

# 4. Implementation

We now describe our testbed and Implementation.

*Environment:* The smartphone used for experiments was an LG Nexus 5X running Android version 8.0, Linux kernel version 3.18.72, on Hexa-core (4x1.4 GHz Cortex-A53 & 2x1.8 GHz Cortex-A57).

## 4.1. Restart Implementation

In Android, applications use the services of the Android Framework (AF) and run on top of the Dalvik virtual machine, which runs on top of a Linux kernel. The AF has an Activity Manager (AM) component, which is in charge of orchestrating app execution, including the transition between activities.

In Figure 3, we show our Implementation. For simplicity, we only depict one running app, but Android runs multiple apps concurrently in practice. Let us assume that the app contains two activities, A and B, and due to an input event, e.g., the user pressing a button, the app wants to transition from A to B. In the standard Implementation of Android, the activity transition will follow the "old pathway" (shown in gray color on top, denoted S1), which will transition directly from A to B. In our Implementation, the transition can follow two new pathways, S3 and S4 (described shortly), that involve an intervening restart, e.g., restart A before the transition, or restart B after the transition.

Restart Strategy: We now describe the four restart strategies, labeled S1–S4, that govern how the system should proceed when transitioning from activity A to activity B: moreover, our approach supports a third new pathway, named S2, where A is restarted even when no transition is necessary, to confuse the attacker. The restarts are carried out by a restart engine, described shortly.

S1: The "old" approach, without restart, where we transition from activity A to activity B (shown in gray, on top of Figure 3).

S2: A restart approach without transition: just restart A (shown in green in Figure 3).

S3: Our main proposed restart approach: restart A, then transition from A to B (shown in red in Figure 3).

S4: An alternative restart approach: transition from A to B, then restart B (shown in blue in Figure 3).

We have implemented and experimented with all these four strategies.

## 4.2. Restart Engine

As shown in Figure 3, the restart engine consists of four modules: Restart Events Injector, Runtime Information Retriever, Runtime Information Analyzer, and Task Hijacking Detector.

The Restart Events Injector carries out the S2–S4 restart strategies via direct event injection. Specifically, it automatically injects keyboard and touch events into the Android OS, so activity transitions follow the new pathways

without user intervention. In Android, applications are driven by various events, e.g., screen touch events and keyboard events. Some events can cause the current activity to stop, pause or be destroyed. For example, pressing the 'Home' button stops the current activity, the 'Back' button destroys the current activity, and touching the menu can pause the current activity. By injecting these events, our restart engine can restart activities at all three levels, 1–3, described in Section II-A:

Level 1 – Pause: To restart at pause level, we inject a pair of events to partially obscure the activity and then recover the visibility of the activity.

Level 2 – Stop: To restart at stop level, we inject a pair of events to obscure the activity and then recover fully the visibility of the activity.

Level 3 – Destroy: To restart at destroying level, we inject a pair of events to kill the activity and then create the activity again.

In order to inject the pair of events, the restart engine directly writes events to /dev/input/eventX.

Our previous Implementation [6] used AM services and could only restart at level 2, but our current version can restart at levels 1–3. Moreover, the previous Implementation used an Android shell (ADB commands), but this introduced about 400–500 ms of idle time between restart and the transition, which hinders usability. The /dev/input/eventX are special files designed for event input, part of the Linux kernel underlying Android. To access /dev/input/eventX, the restart engine runs as a native process directly on top of the kernel. Finally, the previous Implementation was a shell script, while the new engine has been rewritten from scratch and consists of about 4,000 lines of code in C and assembly, which runs as a standalone process. As a result, the new engine implementation is much more efficient. Moreover, this separate-process approach avoids heavyweight solutions such as changing the AF which is impractical, as the AF changes subsection from version to version, and would require "rooting" the phone to install the custom AF.

Note that our solution does not affect background services because the restart engine only inserts events for restarting activities without calling stopService() to stop any background services.

The Runtime Information Retriever gets information about the current back stacks and activities from the Activity Manager: because restarts can change back stacks and activities, the Retriever queries the Activity Manager before, during, and after each restart. Then, the Runtime Information Analyzer interprets the runtime information obtained from the Activity Manager and reconstructs back stacks.

We now describe the detection strategy. The Task Hijacking Detector module detects abnormal changes to back stacks, as explained shortly, which can indicate an attack. An alarm is posted to notify the user if an attack is detected. After the foreground activity is destroyed, the activity just beneath the foreground activity should be shown. Otherwise, it is an attack. According to Ren et al. [5], we have designed the rules to recognize Task Hijacking attacks after destroying the foreground activity as follows:

- Showing another activity (e.g., the Home activity) instead of the activity beneath indicates a spoofing attack.
- Creating a new back stack and bringing it to the front indicates a back hijacking attack or phishing attack.
- Destroying all activities except the root activity of the stack and then showing the root activity indicate a malware uninstall prevention attack.
- Still showing the current activity from a different application compared to that of root activity indicates a Ransomware attack.

## 5. EVALUATION

We now present our evaluation. First, we provide an overview of the apps and app selection process. Then we discuss the experimental methodology and the results.

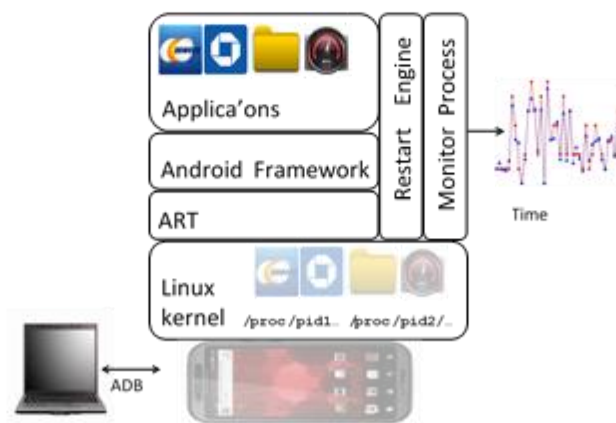



Figure 4: Overview of our data collection process.

### *5.1 Examined Apps*

For evaluation, we chose 43 activity transitions in 34 Android apps. We used several criteria when selecting the apps to ensure a representative sample: apps had to be popular, spanning free and paid categories (built-in and third-party categories); and have a wide range of sizes. Table II presents the apps: name, popularity (number of installs per Google Play), and size. Thirty-two apps are free, and 2 are paid (indicated by the $$ sign). Of the 32 free apps, 24 are third-party apps available on Google Play, and 8 are built-in apps that come preinstalled with the phone.

Fourteen third-party apps are very popular, with more than 1 million installs. Moreover, two of them — Chase Mobile and Newegg Mobile — are security-critical since they are used for online banking and shopping; a security attack against them can expose the user's bank account information or credit card numbers. Apps have a range of sizes, from medium (367 KB) to large (30 MB).

### *5.2 Data Collection*

The data collection process is shown in Figure 4. The restart engine triggers restarts and, for efficiency reasons, runs as a native Linux process rather than a VM-based app. The monitor process collects experimental data and monitors app execution. The test phone is connected to a laptop via the Android Debugging Bridge (ADB). The monitor process sends data through ADB to the laptop. The monitor process takes a sample every 8 milliseconds and collects side-channel information. In particular, it samples the third entry in /proc/pid/statm of an application under test and outputs a sequence of samples that constitute the time series. We then process the samples using time series analysis, as explained shortly.

TABLE 2
Test apps characteristics

| App | Popularity (# installs) | Size (KB) |
|---|---|---|
| Facebook Mobile | 1,000,000,000+ | 3,384 |
| WeChat | 100,000,000+ | 30,970 |
| FileExplorer | 100,000,000+ | 9,315 |
| Chase Mobile | 10,000,000+ | 10,362 |
| BBC News | 10,000,000+ | 1,950 |
| FoxFi | 5,000,000+ | 388 |
| Music Pro ($$) | 5,000,000+ | 7,483 |
| 1MobileMarket | 1,000,000+ | 6,717 |
| No-frills CPU Control | 1,000,000+ | 1,100 |

| Phone Copier | 1,000,000+ | 2,259 |
|---|---:|---:|
| VPN Connections | 1,000,000+ | 2,290 |
| Norton Snap | 1,000,000+ | 1,654 |
| NPR News | 1,000,000+ | 543 |
| Newegg Mobile | 1,000,000+ | 9,900 |
| Alarm Klock | 500,000+ | 119 |
| Edge | 100,000+ | 4,768 |
| myControl | 100,000+ | 141 |
| Call Reminder | 50,000+ | 3,804 |
| GPS Test Plus | 50,000+ | 409 |
| AnCal | 10,000+ | 216 |
| AndSiddur | 10,000+ | 645 |
| OpenLiveView | 10,000+ | 455 |
| GBC Emulator ($$) | 10,000+ | 367 |
| HoloKen | 5,000+ | 408 |
| OpenGLDemo | 5,000+ | 10,012 |
| HoloConvert | 1,000+ | 937 |
| Gallery3d | (builtin) | 5,122 |
| VideoEditor | (builtin) | 5,243 |
| Calendar | (builtin) | 1,751 |
| DeskClock | (builtin) | 2,311 |
| People | (builtin) | 4,634 |
| Music | (builtin) | 19,148 |
| Browser | (builtin) | 2,536 |
| CalenMob | (builtin) | 5,622 |

.

### *5.3 Resistance to Known Attacks: Activity Inference*

According to Chen et al. [4], we constructed the Activity Inference attacks. The critical step of the attack is to detect activity transitions. If we prevent that detection, we prevent the attack. According to Chen et al. [4], activity transitions are detected based on shared_vm events and idle times between events. The critical step in recognizing an activity transition period is identifying the idle time between two transitions. According to our observation, the length of idle time between transitions is more than 1,200 ms if the transitions are triggered manually by a user. Therefore, we have chosen 1,200 ms as the idle time threshold to separate consecutive transitions.

The defense results of all original transitions (i.e., S1) and combined transitions (i.e., S3 and S4) are listed in Table III. 29 out of 30 original transitions were detected; hence the attack succeeded. This result is similar to the results of

Table 3:

Defending against Activity Inference attacks (Fail/Succeed).

| App | Activity Transitions | No | Stop-level | | Destroy-level | |
|---|---|---|---|---|---|---|
| | | Restart | Restart | | Restart | |
| | | S1 | S3 | S4 | S3 | S4 |
| Chase Mobile | Home → PrivacyOptions | F | S | S | S | S |
| | Home → FindBranch | F | S | S | S | S |
| Newegg Mobile | Main → ShoppingCart | F | S | S | S | S |
| | Main → WishListItem | F | S | S | S | S |
| | Main → Login | F | S | S | S | S |
| | Main → OrderHistory | F | S | S | S | S |

| | | | | | | |
|---|---|---|---|---|---|---|
| | Main → MyPersonalHomeCust | F | S | S | S | S |
| Browser | BrowserActivity → BrowserPrefsPg | F | S | S | S | S |
| GBC Emulator($$) | MainActivity → EmulatorSettings | F | S | S | S | S |
| Gallery3d | app.Gallery → stngs.GallerySttgs | F | S | S | S | S |
| VideoEditor | ProjectsActivity → VideoEditorActivity | F | S | S | S | S |
| Calendar | AllInOneActivity → CalendarSttgs | S | S | S | S | S |
| | AllInOneActivity → EditEvent | F | S | S | S | S |
| | AllInOneActivity → EventInfo | F | S | S | S | S |
| DeskClock | DeskClock → SettingsActivity | F | S | S | S | S |
| 1MobileMarket | MainActivity → MyAppsInstalledActivity | F | S | S | S | S |
| | MainActivity → SettingsActivity | F | S | S | S | S |
| Facebook Mobile | FacebookLoginActivity → SimpleAccountRegistrationActivity | F | S | S | S | S |
| WeChat | SnsTimeLineUI → SnsUserUI | F | S | S | S | S |
| BBC News | HomeWwActivity → ArticleActivity | F | S | S | S | S |
| No-frills CPU Control | Main → Preferences | F | S | S | S | S |
| | Main → About | F | S | S | S | S |
| Call Reminder | EditorActivity → Contacts | F | S | S | S | S |
| CalenMob | CalenListActivity → EditEventActivity | F | S | S | S | S |
| Alarm Klock | ActivityAlarmClock → ActivityAppSettings | F | S | S | S | S |
| AnCal | AnCal → ActivityTask | F | S | S | S | S |
| AndSiddur | AndSiddurSplash → AndSiddurDaaven | F | S | S | S | S |
| Edge | EdgeSettings → ChooseShortcutsActivity | F | S | S | S | S |
| FileExplorer | Main → ExternalStorage | F | S | S | S | S |
| FoxFi | HotspotSettings → RegisterActivity | F | S | S | S | S |
| GPS Test Plus | GPSTestPlus → AppSettings | F | S | S | S | S |
| HoloConvert | SettingsActivity → MainActivity | F | S | S | S | S |
| HoloKen | MainActivity → SettingsActivity | F | S | S | S | S |
| Music | PlaylistBrowserActivity → MusicBrowserActivity | F | S | S | S | S |
| Music Pro | MainMenu → AutoHarp | F | S | S | S | S |
| myControl | Preferences → myControl | F | S | S | S | S |
| Norton Snap | CaptureActivity → main | F | S | S | S | S |
| NPR News | NewsStoryActivity → NewsListActivity | F | S | S | S | S |
| OpenGLDemo | Main → OpenGLColouredPyramidActivity | F | S | S | S | S |
| OpenLiveView | ConfigWizardActivity → BluetoothSettingsActivity | F | S | S | S | S |
| People | PeopleActivity → ContactEditorActivity | F | S | S | S | S |

| Phone Copier | PbapInfoActivity → AndroidConnectorNewActivity | F | S | S | S | S |
|---|---|---|---|---|---|---|
| VPN Connections | EditNetwork → VPNC | F | S | S | S | S |

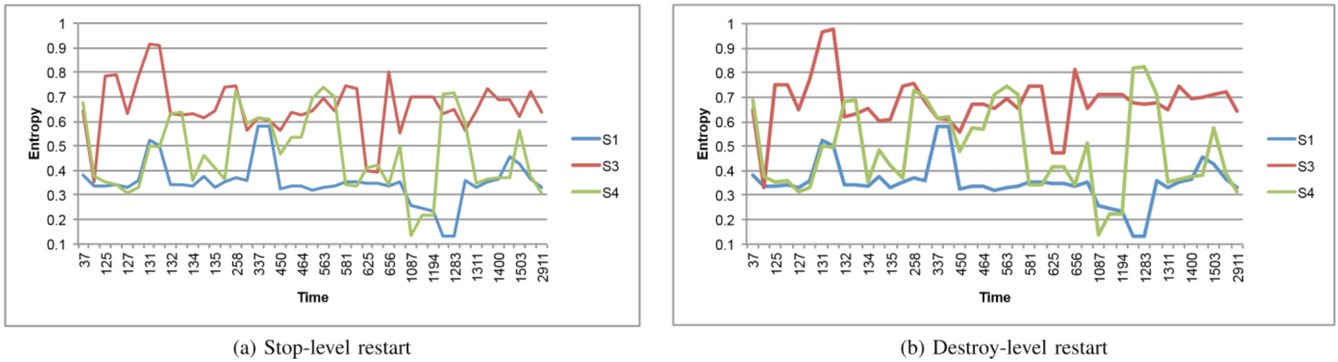
Chen et al. [4]. The missed transition is between activities
CalendarSttgs and AllInOneActivity in the Calendar app. The maximum idle time between events in this transition is greater than 1,200 ms, so the transition is divided into two by the attack tool and thus cannot be recognized.

All of the combined transitions introduced by our scheme successfully withstood the attack, i.e., the attack tool failed to detect activity transitions. The major reason is that the idle time between transition and restart is reduced from more than 1,200 ms to about 400 ms by our automatic restart mechanism. Consequently, the attack tool cannot differentiate between activity transition and restart by just measuring idle time. Note that, to differentiate transitions from restarts, we can reduce the threshold idle time to about 400 ms, reducing the detection rate on the original transitions. A transition consists of a series of shared_vm change events where two consecutive events are separated by an idle period. Idle periods can be 0–1,200 ms. Suppose the threshold idle time is shorter than 1,200 ms. In that case, the attack tool could have a good chance to separate a single activity into two by considering an idle period within a single transition as an idle threshold period between two different transitions.

In addition, our approach makes restart event detection more difficult because our solution combines a restart with a transition. The restart and the transition consist of a series of shared-memory change events – it is difficult to separate the restart from the events of transition. This is because the idle time between restart and transition falls into the range of the idle time of two consecutive events. Finally, the idle time between events or between restart and transition can change across different runs.

TABLE 4:
Defending against Task Hijacking attacks (Fail/Succeed).

| Attack Types | Restart Strategies | Newegg Mobile | Facebook Mobile | GPS Test Plus | FoxFi | Settings | VPN Connections |
|---|---|---|---|---|---|---|---|
| | | Login | FacebookLoginActivity | AppSettings | RegisterActivity | SubSettings | VPNC |
| Spoofing Attack | S1 | F | F | F | F | F | F |
| | S2 | S | S | S | S | S | S |
| | S3 | S | S | S | S | S | S |
| | S4 | S | S | S | S | S | S |
| Phishing Hijacking | S1 | F | F | F | F | F | F |
| | S2 | S | S | S | S | S | S |
| | S3 | S | S | S | S | S | S |
| | S4 | S | S | S | S | S | S |
| Preventing Uninstallation | S1 | - | - | - | - | F | - |
| | S2 | - | - | - | - | S | - |
| | S3 | - | - | - | - | S | - |
| | S4 | - | - | - | - | S | - |
| Ransomware | S1 | F | F | F | F | F | F |
| | S2 | S | S | S | S | S | S |
| | S3 | S | S | S | S | S | S |
| | S4 | S | S | S | S | S | S |




(a) Stop-level restart

(b) Destroy-level restart

Figure 5: Entropy v. transition time (in ms).

### *5.4. Resistance to Known Attacks: Task Hijacking*

We constructed the Task Hijacking attacks according to Ren et al. [5]. There are four types of Task Hijacking attacks: ransomware, spoofing attack, phishing-hijacking, and preventing uninstallation. For each type of attack, we created a malicious app that can hijack other apps by manipulating task state transition conditions ( taskAffinity and allowTaskReparenting as described in Section II-C). Then we used the malicious apps to successfully hijack six apps with the four kinds of attacks. The attack of preventing uninstallation was only used to hijack Android Settings, the major method to uninstall an app from the Android system.

The detection results are shown in Table IV. For the restart strategies of S2, S3 and S4, our mechanism can detect the four kinds of Task Hijacking attacks as the corresponding activities are restated at destroying level. The attacks cannot be detected for the restart strategy of S1 that is the original activity transition without any restart. This means that restart can help detect Task Hijacking attacks.

### *5.5. Resistance to Unknown Attacks*

We now quantify the potential effectiveness of our approach against *unknown attacks* that use shared memory as a side channel. We use time series complexity as an effectiveness measure, as explained next.

*Time series complexity:* Recall that attacks rely on predictability of app behavior as reflected in the /proc/pid/statm time series values: if the time series has high predictability (aka low complexity), the attack has a high chance of success. If the time series has low predictability (aka high complexity), the attacker will have difficulty inferring app behavior. To measure time series complexity, we use the well-known *permutation entropy* (*PE*) metric [7] normalized so that $0 \le PE \le 1$. Here 0 represents no entropy, while 1 represents a random time series. Hence higher PE values are more desirable.

*Time series results:* In Figure 6 we plot the entropy as a function of transition time for strategies S1, S3, and S4, for restarts at both Stop and Destroy levels. Individual results will be presented and discussed shortly. The superiority of S3 (red series) is clear: it is consistently higher (by 86–92.7% on average) than S1 (blue series). Moreover, it is consistently high throughout the transition time range, from 37 ms to 2911 ms. The effectiveness of S4 (green series), while better than S1 by 32.7–38.2%, is lower than S3 and not as consistent – for certain transitions, S4 has lower entropy than S1. Therefore, S3 is the preferred approach.

Table 5:
Evaluation results of restarting at *Stop* level. Reported values are averages of ten runs.

| App | Activity Transitions | Permutation Entropy | | | | Transition Time (ms) | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | S1 | S2 | S3 | S4 | S1 | S2 | S3 | S4 |
| Chase Mobile | Home → PrivacyOptions | 0.331 | 0.361 | 0.691 | 0.741 | 563 | 196 | 761 | 755 |
| | Home → FindBranch | 0.334 | 0.238 | 0.804 | 0.342 | 656 | 189 | 854 | 842 |
| Newegg Mobile | Main → ShoppingCart | 0.257 | 0.408 | 0.701 | 0.135 | 1087 | 164 | 1256 | 1253 |
| | Main → WishListItem | 0.364 | 0.328 | 0.723 | 0.378 | 1521 | 63 | 1590 | 1584 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Main → Login | 0.367 | 0.374 | 0.687 | 0.369 | 1400 | 168 | 1578 | 1563 |
| | Main → OrderHistory | 0.457 | 0.352 | 0.686 | 0.373 | 1460 | 85 | 1553 | 1541 |
| | Main → MyPersonalHomeCust | 0.354 | 0.333 | 0.732 | 0.364 | 1359 | 50 | 1417 | 1406 |
| Browser | BrowserActivity → BrowserPrefsPg | 0.344 | 0.342 | 0.623 | 0.635 | 134 | 19 | 154 | 150 |
| GBC Emulator ($$) | MainActivity → EmulatorSettings | 0.578 | 0.352 | 0.604 | 0.607 | 343 | 95 | 444 | 440 |
| Gallery3d | app.Gallery → stngs.GallerySttgs | 0.523 | 0.395 | 0.917 | 0.499 | 131 | 63 | 204 | 194 |
| VideoEditor | ProjectsActivity → VideoEditorActivity | 0.339 | 0.405 | 0.785 | 0.351 | 125 | 99 | 233 | 218 |
| Calendar | AllInOneActivity → CalendarSttgs | 0.333 | 0.337 | 0.637 | 0.307 | 2911 | 203 | 3116 | 3114 |
| | AllInOneActivity → EditEvent | 0.353 | 0.348 | 0.735 | 0.337 | 586 | 140 | 734 | 729 |
| | AllInOneActivity → EventInfo | 0.335 | 0.344 | 0.645 | 0.701 | 564 | 83 | 653 | 648 |
| DeskClock | DeskClock → SettingsActivity | 0.358 | 0.335 | 0.565 | 0.593 | 1307 | 248 | 1557 | 1552 |
| 1MobileMarket | MainActivity → MyAppsInstalledActivity | 0.131 | 0.336 | 0.648 | 0.718 | 1283 | 73 | 1364 | 1355 |
| | MainActivity → SettingsActivity | 0.331 | 0.332 | 0.643 | 0.348 | 1311 | 137 | 1455 | 1443 |
| Facebook Mobile | FacebookLoginActivity → SimpleAccountRegistrationActivity | 0.335 | 0.306 | 0.635 | 0.535 | 453 | 109 | 566 | 559 |
| WeChat | SnsTimeLineUI → SnsUserUI | 0.236 | 0.239 | 0.702 | 0.218 | 1194 | 140 | 1340 | 1335 |
| BBC News | HomeWwActivity → ArticleActivity | 0.346 | 0.359 | 0.398 | 0.413 | 625 | 88 | 721 | 711 |
| No-frills CPU Control | Main → Preferences | 0.358 | 0.333 | 0.784 | 0.329 | 128 | 89 | 215 | 215 |
| | Main → About | 0.334 | 0.337 | 0.353 | 0.375 | 120 | 6 | 125 | 125 |
| Call Reminder | EditorActivity → Contacts | 0.354 | 0.332 | 0.737 | 0.367 | 164 | 32 | 201 | 197 |
| CalenMob | CalenListActivity → EditEventActivity | 0.345 | 0.346 | 0.63 | 0.634 | 132 | 19 | 150 | 145 |
| Alarm Klock | ActivityAlarmClock → ActivityAppSettings | 0.583 | 0.355 | 0.615 | 0.613 | 337 | 89 | 435 | 420 |
| Music Pro | MainMenu → AutoHarp | 0.132 | 0.339 | 0.63 | 0.709 | 1271 | 40 | 1312 | 1311 |
| myControl | Preferences → myControl | 0.338 | 0.329 | 0.633 | 0.361 | 134 | 36 | 176 | 173 |
| Norton Snap | CaptureActivity → main | 0.376 | 0.344 | 0.614 | 0.46 | 135 | 34 | 166 | 168 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| NPR News | NewsStoryActivity → NewsListActivity | 0.43 | 0.411 | 0.62 | 0.561 | 1503 | 122 | 1631 | 1626 |
| OpenGLDemo | Main → OpenGLColouredPyramidActivity | 0.339 | 0.302 | 0.628 | 0.537 | 464 | 65 | 532 | 522 |
| OpenLiveView | ConfigWizardActivity → BluetoothSettingsActivity | 0.352 | 0.39 | 0.552 | 0.502 | 982 | 117 | 1100 | 1098 |
| People | PeopleActivity → ContactEditorActivity | 0.326 | 0.32 | 0.562 | 0.465 | 450 | 117 | 573 | 566 |
| Phone Copier | PbapInfoActivity → AndroidConnectorNewActivity | 0.244 | 0.234 | 0.699 | 0.217 | 1192 | 89 | 1290 | 1282 |
| AnCal | AnCal → ActivityTask | 0.33 | 0.37 | 0.643 | 0.41 | 135 | 36 | 177 | 166 |
| AndSiddur | AndSiddurSplash → AndSiddurDaaven | 0.383 | 0.335 | 0.644 | 0.675 | 37 | 33 | 74 | 72 |
| Edge | EdgeSettings → ChooseShortcutsActivity | 0.499 | 0.387 | 0.912 | 0.494 | 131 | 58 | 186 | 191 |
| FileExplorer | Main → ExternalStorage | 0.341 | 0.406 | 0.791 | 0.345 | 125 | 31 | 154 | 152 |
| FoxFi | HotspotSettings → RegisterActivity | 0.332 | 0.338 | 0.631 | 0.311 | 127 | 44 | 169 | 174 |
| GPS Test Plus | GPSTestPlus → AppSettings | 0.354 | 0.346 | 0.745 | 0.341 | 581 | 76 | 658 | 657 |
| HoloConvert | SettingsActivity → MainActivity | 0.317 | 0.352 | 0.642 | 0.693 | 555 | 79 | 638 | 636 |
| HoloKen | MainActivity → SettingsActivity | 0.361 | 0.343 | 0.563 | 0.594 | 312 | 31 | 353 | 344 |
| Music | PlaylistBrowserActivity → MusicBrowserActivity | 0.37 | 0.336 | 0.747 | 0.725 | 258 | 36 | 294 | 290 |
| VPN Connections | EditNetwork → VPNC | 0.347 | 0.353 | 0.394 | 0.421 | 626 | 9 | 637 | 638 |
| *Average* | | 0.352 | 0.343 | 0.659 | 0.468 | 672 | 86 | 763 | 757 |
| compared with S1 | | | -0.009 | +0.307 | +0.116 | | -586 | +91 | +85 |
| | | | *-2.6%* | *+87%* | *+32.7%* | | *-90%* | *+13.4%* | *+12.6%* |

Table 5 shows the individual results, for each transition, of restarting at stop level. Columns 3–6 are the results of the entropy measures. The main comparison is between strategy S1, i.e., the default Android implementation, and S3 (our main approach). Note how the PE is consistently higher in S3 than in S1. The "Average" row (third from bottom) shows the average values across all activities. The second-to-last row show the S2-S1, S3-S1, and S4-S1 differences, respectively. The last row shows the same difference in percent, compared to S1. We make several observations.

Note how PE increases from 0.352 on average (S1) to 0.659 (S3) – an 87% increase, demonstrating that our proactive restart approach is effective at introducing randomness in the time series and consequently is effective at making the attacker's job harder. Strategies S2 and S4 are less effective if used in isolation (though S4 has a 32.7% higher PE than S1).

Table 6 shows the results of restarting at destroying level; the Table has the same structure as Table V. Note that this strategy is *even more effective* than stop-level restarts: the average PE increases from 0.352 (S1) to 0.679 (S3) – a 92.7% increase. Strategies S2 and S4 are less effective if they are used in isolation (though S4 has a 38.2% higher PE compared wi S1). Finally, we found only one case out of 86 transitions where S3 fails to outperform S1 (app No-frills CPU Control transition Main → About, destroy-level restart), where S3's entropy is 0.331 while S1's is 0.334.

TABLE 6:
Evaluation results of restarting at *Destroy* level. Reported values are averages of ten runs.

| App | Activity Transitions | Permutation Entropy | | | | Transition Time (ms) | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | S1 | S2 | S3 | S4 | S1 | S2 | S3 | S4 |
| Chase Mobile | Home → PrivacyOptions | 0.331 | 0.369 | 0.696 | 0.745 | 563 | 215 | 787 | 775 |
| | Home → FindBranch | 0.334 | 0.239 | 0.812 | 0.344 | 656 | 215 | 868 | 873 |
| Newegg Mobile | Main → ShoppingCart | 0.257 | 0.404 | 0.708 | 0.137 | 1087 | 160 | 1247 | 1247 |
| | Main → WishListItem | 0.364 | 0.332 | 0.722 | 0.386 | 1521 | 79 | 1603 | 1594 |
| | Main → Login | 0.367 | 0.374 | 0.691 | 0.377 | 1400 | 166 | 1571 | 1560 |
| | Main → order history | 0.457 | 0.35 | 0.699 | 0.38 | 1460 | 101 | 1567 | 1564 |
| | Main → MyPersonalHomeCust | 0.354 | 0.33 | 0.745 | 0.366 | 1359 | 66 | 1432 | 1426 |
| Browser | BrowserActivity → BrowserPrefsPg | 0.344 | 0.341 | 0.629 | 0.687 | 134 | 21 | 162 | 148 |
| GBC Emulator ($$) | MainActivity → EmulatorSettings | 0.578 | 0.349 | 0.61 | 0.619 | 343 | 122 | 473 | 462 |
| Gallery3d | app.Gallery → stngs.GallerySttgs | 0.523 | 0.397 | 0.967 | 0.503 | 131 | 79 | 220 | 213 |
| VideoEditor | ProjectsActivity → VideoEditorActivity | 0.339 | 0.403 | 0.753 | 0.353 | 125 | 133 | 266 | 255 |
| Calendar | AllInOneActivity → CalendarSttgs | 0.333 | 0.333 | 0.645 | 0.312 | 2911 | 221 | 3130 | 3132 |
| | AllInOneActivity → EditEvent | 0.353 | 0.344 | 0.745 | 0.342 | 586 | 166 | 753 | 746 |
| | AllInOneActivity → EventInfo | 0.335 | 0.344 | 0.653 | 0.713 | 564 | 108 | 682 | 670 |
| DeskClock | DeskClock → SettingsActivity | 0.358 | 0.336 | 0.677 | 0.71 | 1307 | 276 | 1582 | 1577 |
| 1MobileMarket | MainActivity → MyAppsInstalledActivity | 0.131 | 0.329 | 0.671 | 0.827 | 1283 | 61 | 1352 | 1339 |
| | MainActivity → SettingsActivity | 0.331 | 0.33 | 0.651 | 0.356 | 1311 | 166 | 1477 | 1477 |
| Facebook Mobile | FacebookLoginActivity → SimpleAccountRegistrationActivity | 0.335 | 0.302 | 0.672 | 0.572 | 453 | 83 | 543 | 531 |
| WeChat | SnsTimeLineUI → SnsUserUI | 0.236 | 0.233 | 0.71 | 0.222 | 1194 | 115 | 1313 | 1310 |
| BBC News | HomeWwActivity → ArticleActivity | 0.346 | 0.464 | 0.473 | 0.416 | 625 | 7 | 638 | 629 |
| No-frills CPU Control | Main → Preferences | 0.358 | 0.329 | 0.776 | 0.33 | 128 | 116 | 254 | 243 |
| | Main → About | 0.334 | 0.332 | 0.331 | 0.377 | 120 | 22 | 143 | 142 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Call Reminder | EditorActivity → Contacts | 0.354 | 0.338 | 0.746 | 0.371 | 164 | 41 | 208 | 200 |
| CalenMob | CalenListActivity → EditEventActivity | 0.345 | 0.331 | 0.619 | 0.681 | 132 | 25 | 167 | 150 |
| Alarm Klock | ActivityAlarmClock → ActivityAppSettings | 0.583 | 0.358 | 0.617 | 0.613 | 337 | 120 | 463 | 453 |
| Music Pro | MainMenu → AutoHarp | 0.132 | 0.334 | 0.677 | 0.817 | 1271 | 64 | 1342 | 1336 |
| myControl | Preferences → myControl | 0.338 | 0.331 | 0.654 | 0.356 | 134 | 43 | 180 | 178 |
| Norton Snap | CaptureActivity → main | 0.376 | 0.334 | 0.604 | 0.483 | 135 | 46 | 186 | 182 |
| NPR News | NewsStoryActivity → NewsListActivity | 0.43 | 0.448 | 0.709 | 0.575 | 1503 | 152 | 1654 | 1654 |
| OpenGLDemo | Main → OpenGLColouredPyramidActivity | 0.339 | 0.3 | 0.671 | 0.57 | 464 | 86 | 560 | 550 |
| OpenLiveView | ConfigWizardActivity → BluetoothSettingsActivity | 0.352 | 0.44 | 0.655 | 0.514 | 982 | 140 | 1130 | 1118 |
| People | PeopleActivity → ContactEditorActivity | 0.326 | 0.339 | 0.559 | 0.478 | 450 | 151 | 610 | 597 |
| Phone Copier | PbapInfoActivity → AndroidConnectorNewActivity | 0.244 | 0.231 | 0.713 | 0.224 | 1192 | 114 | 1311 | 1301 |
| AnCal | AnCal → ActivityTask | 0.33 | 0.372 | 0.611 | 0.423 | 135 | 50 | 193 | 185 |
| AndSiddur | AndSiddurSplash → AndSiddurDaaven | 0.383 | 0.325 | 0.65 | 0.688 | 37 | 58 | 103 | 94 |
| Edge | EdgeSettings → ChooseShortcutsActivity | 0.499 | 0.392 | 0.978 | 0.498 | 131 | 88 | 221 | 212 |
| FileExplorer | Main → ExternalStorage | 0.341 | 0.4 | 0.751 | 0.357 | 125 | 51 | 182 | 174 |
| FoxFi | HotspotSettings → RegisterActivity | 0.332 | 0.321 | 0.646 | 0.315 | 127 | 68 | 199 | 190 |
| GPS Test Plus | GPSTestPlus → AppSettings | 0.354 | 0.347 | 0.744 | 0.345 | 581 | 90 | 668 | 672 |
| HoloConvert | SettingsActivity → MainActivity | 0.317 | 0.344 | 0.652 | 0.71 | 555 | 105 | 670 | 660 |
| HoloKen | MainActivity → SettingsActivity | 0.361 | 0.337 | 0.681 | 0.701 | 312 | 46 | 365 | 354 |
| Music | PlaylistBrowserActivity → MusicBrowserActivity | 0.37 | 0.335 | 0.755 | 0.727 | 258 | 44 | 300 | 300 |
| VPN Connections | EditNetwork → VPNC | 0.347 | 0.469 | 0.471 | 0.415 | 626 | 8 | 634 | 632 |
| *Average* | | 0.352 | 0.349 | 0.679 | 0.487 | 672 | 99 | 777 | 769 |
| compared with S1 | | | -0.003<br>*-1.1%* | +0.327<br>*+92.7%* | +0.135<br>*+38.2%* | | -573<br>*-90%* | +105<br>*+15.6%* | +97<br>*+14.5%* |

We have also tested restarting at pause level (level 1), but the results showed that it does not invoke any extra shared_vm change events. Therefore, restarting at pause level cannot increase the PE value; consequently, *level-1 restart is not a good attack mitigation strategy*.

### *5.6. Efficiency*

We now quantify the efficiency of our approach: are transitions taking longer with restart, and if so, how much longer?

The performance data of restarting at the stop level is shown in the last four columns of Table V. The transition time is from the time the transition was initiated to when it has been completed (including restart time) for each of the four strategies. Compared with S1, S3 increases transition time by 91 ms, i.e., a 13.4% increase. Strategy S2 takes less time than S1 since no transition is involved. Strategy S4 increases transition time by 85 ms, a 12.6% increase compared with S1.

The performance data of restarting at destroy level is shown in the last four columns of Table VI. Compared with S1, S3 increases transition time by 105 ms, i.e., a 15.6% increase. Strategy S4 increases transition time by 97 ms, a 14.5% increase compared with S1.

According to human-computer interaction research, visual feedback should not be delayed more than 100 ms lest the user perceives the lag [8], [9]. This exposes the cost-benefit tradeoff: we pay the price of increasing transition time by 85–105 ms for the benefit of increasing attack resilience. Our system fulfills this desideratum; in fact our Implementation could be optimized further to further reduce transition time, but we believe the current Implementation demonstrates that our research approach is viable and effective.

An interesting question is whether transition time can be used as a side-channel. The answer depends on a specific app's distribution of activity transition times. First, note that there is "natural variability" in transition time, which gives us an envelope for small random pauses to counter this side channel. There is a 6–8% variation in no-restart transition time, for the same activity. Specifically, we calculated the coefficient of variation of S1 (no restart) transition time across the 10 runs for each transition; the result was 0.06–0.08. The S1 transition time can vary roughly by 3–232 ms across runs for a certain app and a certain transition. We investigated this further and found that activity transition times can be affected by many factors, e.g., other apps running on the same phone, Android OS version and configuration, network protocol and bandwidth, memory size, CPU frequency. Hence we could use random sleeps on top of restart to "pad" the transition time and counter this side channel. Second, a different-in-scope investigation would be required to study the distribution of no-restart transition time to make a conclusive statement about transition time as a side-channel – such an investigation is beyond the scope of this paper. In particular, we would need to find apps that permit a large number of different activity transitions, e.g., 20 or 30 such transitions per app so we can study their distribution. For our investigated apps the results are inconclusive: for Newegg and Calendar, out of N studied transitions, N-1 have similar transition times while the remaining sample is an outlier (e.g., 1359–1521 vs. 1087 for Newegg, and 564–586 vs. 2911 for Calendar). Certainly, our approach can mask the inter-variability within the N-1 sample set. To conclude, while transition time could be used as a side channel (in those specific cases where transition times are well separated across different transitions, and the inter-run variability for the same transition is low) this is orthogonal to our approach.

### *5.7 Efficiency vs. Effectiveness Tradeoff*

We now discuss the tradeoff between added security and added overhead (time) when employing our approach. In Figure 6 we plot the increase-in-entropy on the *y*-axis vs increase-in-time on the *x*-axis for S3 (top) and S4 (bottom), when compared with stock Android, S1. For example, the upper-left point in Figure 6b shows that our approach increase entropy by a factor of 4.12x (412%) while increasing transition time by 0.053x (5.3%) compared to S1. The red line is the 45-degree (diagonal) reference line.

The top, S3, charts allow us to make the crucial observation that our approach provides a good security vs. overhead tradeoff: most of the S3 points are located *above the diagonal*, in the 50%–200% range for *y* and 0%–50% range for *x*, which means S3 increases entropy *more* than it increases transition time. Recall, though, that even "high" increases in transition time, e.g., 100%, are tolerable: as discussed in Section V-F, the additional transition time is only 85– 105 ms on average across all transitions.

The bottom charts illustrate that S4 provides a far worse security vs. overhead tradeoff.

These charts, and the individual data points in Tables Table V and Table VI can be used to guide users with deployment of our scheme, e.g., in environments where security is paramount and delays are tolerable, S3 can be deployed for all activities; whereas in environments where security is not crucial, users might chose to only protect those activities whose points fall above the diagonal, where the security benefit surpasses the overhead.

Finally, note that our approach *does not impose any steadystate overhead*, since restarts are only performed around activity transitions, not while the user is interacting with the activity (which is where the bulk of the time is spent).

## 6. RELATED WORK

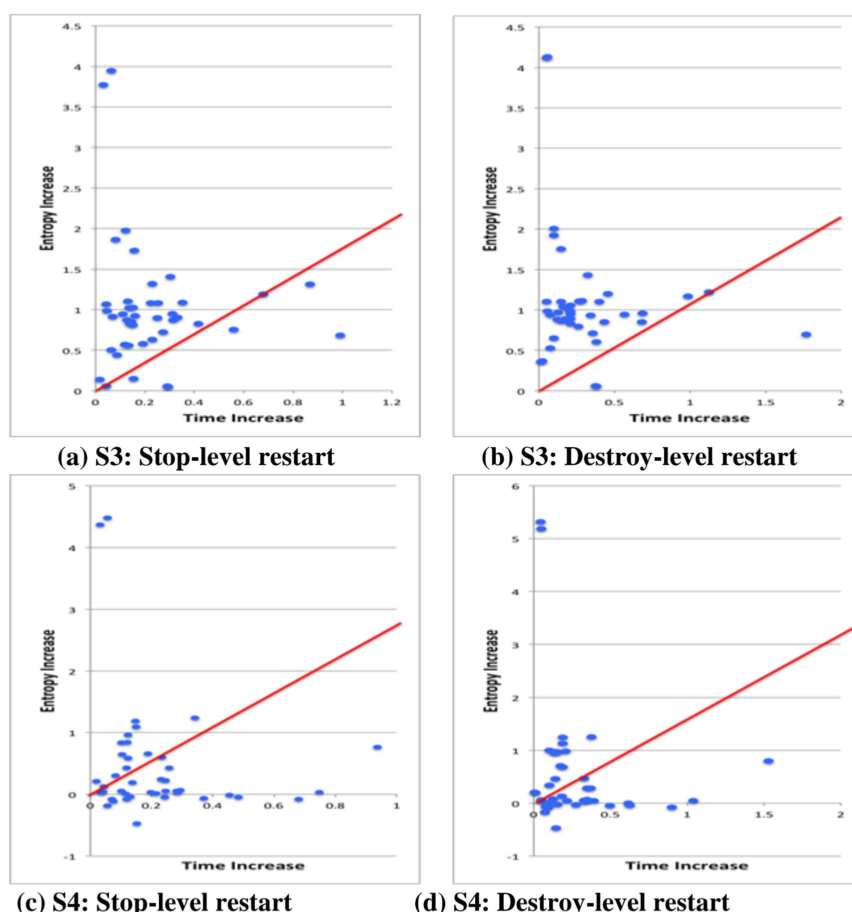
Application Restarts. Application restarts have been used in the past to remedy transient faults. But we are not aware of any work that uses restart as a cyber maneuver to defend against attacks.

Xiang et al. [10] propose proactive rejuvenation strategies to combat software aging problem, which manually or automatically restarts an application or a device. Cotroneo et al. [11] propose a configurable micro-rejuvenation technique to counteract software aging in Android-based mobile devices. Qiao et al. [12] propose a two-level software rejuvenation, with the two levels referring to software applications and the

OS

Perkins et al. [13] used a reactive approach, named ClearView, that monitors an application's execution to learn application invariants, detect bugs or attacks, and automatically construct and apply a patch to heal the application upon detection. ClearView has been applied to Firefox. Ten exploits were presented to ClearView; upon repeated presentation, ClearView learned to identify each exploit and construct a patch against it. Our work is distantly related: our approach is proactive and attack-agnostic, as we do not perform monitoring, detection or patching, whereas ClearView uses sophisticated attack and bug-specific reactive techniques for invariant detection and patch construction.

Sidiroglou et al. [14] developed an approach named ASSURE that employs rescue points to recover from unanticipated failures in desktop/server Linux applications. Candea et al. [15] have proposed "microreboots" (rebooting small components




(a) S3: Stop-level restart (b) S3: Destroy-level restart

(c) S4: Stop-level restart (d) S4: Destroy-level restart

Figure 6: Entropy increase v. transition time increase compared to S1.

instead of entire applications) as a recovery technique for healing capabilities apps recovering from certain classes of Internet services. Our own prior work [16] has used online transient and permanent faults in Android apps. However, patch construction and application restart to provide self- that approach was reactive, rather than proactive, and its goal was fault recovery rather than changing the attack surface. Android Side Channels Attacks and Defenses. Much work has been done on studying side channels. *Proc file systems* have been used for side-channel attacks. Zhang et al. [17] found that the ESP/EIP value can be used to infer keystrokes. Qian et al. [18] have used "sequence-number-dependent" packet counter side channels to infer TCP sequence number. In Memento [19], the memory footprints correlate with the web page the user is visiting. Zhou et al. [20] found 3 Android/Linux

public resources to leak private information about location, disease, etc.. Chen et al. [4] proposed Activity inference attacks that can apply to all Android apps. There are few effective defenses against the types of side-channel attacks. Lately, Zhang et al. have proposed App Guardian to pause all suspicious background processes to stop them from gathering any data about the foreground app [21]. Such defense could be effective; however, it comes with a functionality cost — many background apps will not be able to function as designed. For normal users, that means background apps or services such as antivirus, music play, navigation, and apps handing long time network communication etc., would not run. Family locator apps such as Life360, which allows parents to monitor the whereabouts of their children, would be inoperable. The situation can be even more serious when phones are used in other, mission-critical scenarios. For example, a water or power plant operator uses a background app to monitor the plants' operation; if App Guardian stops the app, an alarm would be missed, which would have serious consequences. Another defense against the GUI state manipulation attacks proposed by Bianchi et al. tries to provide explicit and secure indicators to keep the user informed about which app runs in the foreground at all times [22]. Such defense is tailored to attacks similar to Activity inference. Ning [23] and Azab et al. [24] introduced Samsung KNOX – an effective approach that stops attacks centered around modifying or injecting kernel binaries. KNOX can also stop attacks that involve modifying the system memory layout, e.g., through double memory mapping. However, KNOX cannot stop side-channel attacks as these attacks do not change the kernel or system memory layout.

Starting with Android 7 (Nougat) [25], access to an app's/proc filesystem is restricted to that app, to prevent potential side-channel attacks. There are two issues with this approach, compared to our approach. First, attackers can look for other side channels besides proc, while our approach disrupts system state beyond proc, as shown in Table I. Second, as of March 2017, Nougat's adoption rate is 2.8% [26], which leaves the other 97.2% of phones susceptible to attack. Ren et al. [5] used Task Hijacking to implement UI spoofing attacks. The attacks can manipulate task state transition conditions such that the system instead displays the spoofing UI of malicious activity by relocating the malicious activity from the background task to the top of victim app's back stack. Our method can detect such attacks by restarting the foreground activity at destroying level as presented in Section IV. Compared with Ren et al. [27], our method does not need to intercept Android API functions and thus has zero impact on activity performance.

Our prior work [6] introduced the idea of Android proactive restarts, using time-series entropy as a metric, but contained no evaluation of actual attacks. In this paper we make four major new contributions. First, we have redesigned, rewritten, and optimized the restart engine to effect restarts via direct event injection, which greatly improves the efficiency of our technique by reducing transition time overhead from 74% for S3 (601 ms) and 39% for S4 (317 ms) to 15.6% for S3 (105 ms) and 14.5% for S4 (97 ms), respectively. This lowers the lag introduced by our approach to the human detection threshold of 100 ms [8], [9]. Second, we implement and launch Activity Inference attacks, and measure the effectiveness of our technique against these attacks. Third, we add a new module in the restart engine to detect Task Hijacking attacks; furthermore, we implement and launch Task Hijacking attacks, and measure the effectiveness of our technique against these attacks. Fourth, we implement restarts at destroy and pause levels, in addition to the stop level.

## 6. Conclusion

We have proposed a new security defense approach for Android apps by proactively restarting an app. It can make attackers harder to infer app behavior. Specifically, the approach leverages a smartphone's native support for quick and lossless restarts – an action that is minimally intrusive for users but disruptive and confusing for attackers. We chosed 34 popular Android apps from various domains to evaluate our approach. Results showed that our approach can successfully defend against Activity Inference attacks—pernicious attacks that can be used to phish banking credentials, for instance. Using a time series entropy metric we show that our scheme is effective, as it increases entropy by 87%–92.7% on average, which means the behavior of a victim app is harder to predict, hence quantifiably increasing resilience to unknown attacks that use this side channel. Furthermore, our scheme is efficient, as it increases transition time by only 12.6%–15.6% (which translates to 85–105 ms on average). Note that our approach increases *transition time only*, rather than imposing a steady-state overhead – users spend most of the time *inside an activity* rather than transitioning.

# References


[1] Don Torrieri. Cyber maneuvers and maneuver keys. Proceedings of the 2014 Military Communications Conference, 2014.

[2] Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M. S., Conti, M., & Rajarajan, M. (2014). Android security: a survey of issues, malware penetration, and defenses. IEEE communications surveys & tutorials, 17(2), 998-1022.

[3] McDaniel, P., Jaeger, T., La Porta, T. F., Papernot, N., Walls, R. J., Kott, A., ... & Neamtiu, I. (2014, November). Security and science of agility. In Proceedings of the First ACM Workshop on Moving Target Defense (pp. 13-19).

[4] McDaniel, P., Jaeger, T., La Porta, T. F., Papernot, N., Walls, R. J., Kott, A., ... & Neamtiu, I. (2014, November). Security and science of agility. In Proceedings of the First ACM Workshop on Moving Target Defense (pp. 13-19).

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

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

[7] Shan, Z., Neamtiu, I., Qian, Z., & Torrieri, D. (2015, October). Proactive restart as cyber maneuver for Android. In MILCOM 2015-2015 IEEE Military Communications Conference (pp. 19-24). IEEE.

[8] Bandt, C., & Pompe, B. (2002). Permutation entropy: a natural complexity measure for time series. Physical review letters, 88(17), 174102.

[9] Card, S. K., Robertson, G. G., & Mackinlay, J. D. (1991, March). The information visualizer, an information workspace. In Proceedings of the SIGCHI Conference on Human factors in computing systems (pp. 181-186).

[10] Miller, R. B. (1968, December). Response time in man-computer conversational transactions. In Proceedings of the December 9-11, 1968, fall joint computer conference, part I (pp. 267-277).

[11] Xiang, J., Weng, C., Zhao, D., Tian, J., Xiong, S., Li, L., & Andrzejakb, A. (2018, October). A new software rejuvenation model for Android. In 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) (pp. 293-299). IEEE.

[12] Cotroneo, D., De Simone, L., Natella, R., Pietrantuono, R., & Russo, S. (2022). Software micro-rejuvenation for Android mobile systems. Journal of Systems and Software, 186, 111181.

[13] Qiao, Y., Zheng, Z., Fang, Y., Qin, F., Trivedi, K. S., & Cai, K. Y. (2018). Two-level rejuvenation for android smartphones and its optimization. IEEE Transactions on Reliability, 68(2), 633-652.

[14] Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., ... & Rinard, M. (2009, October). Automatically patching errors in deployed software. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (pp. 87-102).

[15] Sidiroglou, S., Laadan, O., Perez, C., Viennot, N., Nieh, J., & Keromytis, A. D. (2009). Assure: automatic software self-healing using rescue points. ACM SIGARCH Computer Architecture News, 37(1), 37-48.

[16] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., & Fox, A. (2004). Microreboot--a technique for cheap recovery. arXiv preprint cs/0406005.

[17] Azim, M. T., Neamtiu, I., & Marvel, L. M. (2014, September). Towards self-healing smartphone software via automated patching. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (pp. 623-628).

[18] Zhang, K., & Wang, X. (2009, August). Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In USENIX Security Symposium (Vol. 20, p. 23).

[19] Qian, Z., Mao, Z. M., & Xie, Y. (2012, October). Collaborative TCP sequence number inference attack: how to crack sequence number under a second. In Proceedings of the 2012 ACM conference on Computer and communications security (pp. 593-604).

[20] Jana, S., & Shmatikov, V. (2012, May). Memento: Learning secrets from process footprints. In 2012 IEEE Symposium on Security and Privacy (pp. 143-157). IEEE.

[21] Zhou, X., Demetriou, S., He, D., Naveed, M., Pan, X., Wang, X., ... & Nahrstedt, K. (2013, November). Identity, location, disease and more: Inferring your secrets from android public resources. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (pp. 1017-1028).

[22] Zhang, N., Yuan, K., Naveed, M., Zhou, X., & Wang, X. (2015, May). Leave me alone: App-level protection against runtime information gathering on android. In 2015 IEEE Symposium on Security and Privacy (pp. 915-930). IEEE.

[23] Bianchi, A., Corbetta, J., Invernizzi, L., Fratantonio, Y., Kruegel, C., & Vigna, G. (2015, May). What the app is that? deception and countermeasures in the android user interface. In 2015 IEEE Symposium on Security and Privacy (pp. 931-948). IEEE.

[24] Ning, P. (2014, November). Samsung knox and enterprise mobile security. In Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (pp. 1-1).

[25] Azab, A. M., Ning, P., Shah, J., Chen, Q., Bhutkar, R., Ganesh, G., ... & Shen, W. (2014, November). Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (pp. 90-102).

[26] BHARDWAJ, A., PANDEY, K., & CHOPRA, R. Android and iOS Security-An Analysis and Comparison Report.

[27] Ki, T., Park, C. M., Dantu, K., Ko, S. Y., & Ziarek, L. (2019, May). Mimic: UI compatibility testing system for Android apps. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) (pp. 246-256). IEEE.

[28] Ren, C., Liu, P., & Zhu, S. (2017). WindowGuard: Systematic Protection of GUI Security in Android. In NDSS.