Proactive Restart as Cyber Maneuver for Android

Zhiyong Shan Iulian Neamtiu Zhiyun Qian
Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA, 92521
Email: {zhiyong,neamtiu,zhiyunq}@cs.ucr.edu

Don Torrieri
US Army Research Laboratory
Adelphi, MD 20783

***Abstract*—Moving-target defense is an effective strategy for deflecting cyber attacks. The widespread use of smartphones in the tactical field requires novel ways of securing smartphones against an ever-increasing number of zero-day attacks. We propose a new, proactive approach for securing smartphone apps against certain classes of attacks. We leverage smartphone's native support for quick and lossless restarts to make application restart a cyber maneuver meant to deflect and confuse attackers. We propose a time-series entropy metric to quantify attack resilience. We apply our approach to 12 popular Android apps chosen from a variety of domains, including online banking and shopping. Preliminary experiments with using proactive restarts on these apps show that restart is a promising way of increasing attack resilience for a certain class of side-channel attacks named Activity Inference attacks.**

I. Introduction

Moving-target 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)* [15]). Given the diversity of modern attacks and the lack of effective detection mechanisms [9], reactive approaches are becoming less and less viable as defense strategies. In contrast, proactive approaches such as cyber maneuver, in which the defender is continually changing 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 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 [11].

Smartphones are used more and more in the tactical field, e.g., through the Nett Warrior [2] or Android Tactical Assault Kit (ATAK) [1] initiatives. However, with increased smartphone use, the potential for smartphones being the target of attacks also increases. Smartphones used in the tactical field might lack a highly available, trusted channel for downloading and applying security patches which further compounds the problem and suggests 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 key insight is that on smartphones, the high-level application state is naturally preserved across application restarts during which the low-level operating system (OS) state of the application is cleaned up, and most attacks target the OS state. Hence, because *app restart is natural for the application but disruptive for the attacker*, proactive restarts de facto implement CM, protecting the application without significantly degrading its functionality.

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, turns 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 OSs. To provide this first-class support, smartphone OSs automate many state management tasks, e.g., quickly and automatically saving and restoring graphical user interface (GUI) state across restarts, as well as invoking user-defined callbacks for saving and restoring app state. 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 OSs offer little support for them. Given smartphones' first-class support for quick and lossless restart, our key insight is to use restart as an CM strategy with modest costs.

The Android platform and apps have been subjected to a variety of security attacks [9], and a variety of defenses have been proposed. Many classes of attacks rely on observing the victim app for a period of time, to infer its behavior or learn operational parameters that facilitate launching the attack. For example, in an Activity Inference attack [8], 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 classes of attacks will be rendered ineffective.

Drawing on these observations, 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-

Level	Cause
1: Pause activity	Activity becomes (partially) covered; Turn off screen
2: Stop activity	Switch to another app; Start a new activity in the same app; Receive a phone call; Press 'Home' button
3: Destroy activity	Press 'Back' button; Kill app

TABLE I: Android restart levels.

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; 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 first provide details on pause/resume and stop/restart operations on the Android platform (Section II-A). Next (Section II-B) we discuss a recently discovered class of side-channel attacks against Android apps named Activity Inference attacks, where malicious apps can infer the screen contents of a victim app and then preempt the victim app as well as masquerade as the victim app. If the victim app maneuvers such that this side-channel information is distorted or unavailable – as we do – an attack is less likely to succeed.

In Section III we provide a concrete example, the popular Newegg app, of how an Activity Inference attack proceeds and how our approach can hinder such attacks.

In Section IV we discuss our implementation. We use Android's native restart capabilities, as well as app management services, to implement our proactive restart scheme.

In Section V we evaluate our approach on 12 popular Android apps. Using a time series entropy metric we show that our scheme is effective, as it increases entropy by 85% on average, which means the behavior of a victim app is harder to predict. We found that our scheme is efficient, as it imposes an average transition time overhead of 601 msec.

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 12 popular Android apps, quantifying the effectiveness vs. efficiency trade-off.

II. Background

We now present background information on proactive security, as well as the resume/restart mechanism on Android.

A. Android Restart

The Android smartphone platform consists of apps, usually written in Java, running on top of Dalvik, a Java virtual machine, which in turn runs on top of a smartphone-specific Linux kernel. Android apps, due to the nature of the platform, 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. When the user returns to the app, the app is resumed or restarted. Hence, smartphone apps and OSs are designed from the ground up to support pause/resume operations smoothly and efficiently.

In Android, our target platform, there are three main levels of restart. We present these levels in Table I: the level is on the left, and the cause is on the right. A restart cycle has little impact on the app and app state: the OS automatically saves and restores GUI state. However, at the OS level, a restart cycle (especially at level 3 — destroy app) is very disruptive, as the process is killed. When the app restarts, it restarts with different OS state, e.g., process identifier (PID), memory mapping, process counters from `/proc` files, etc.

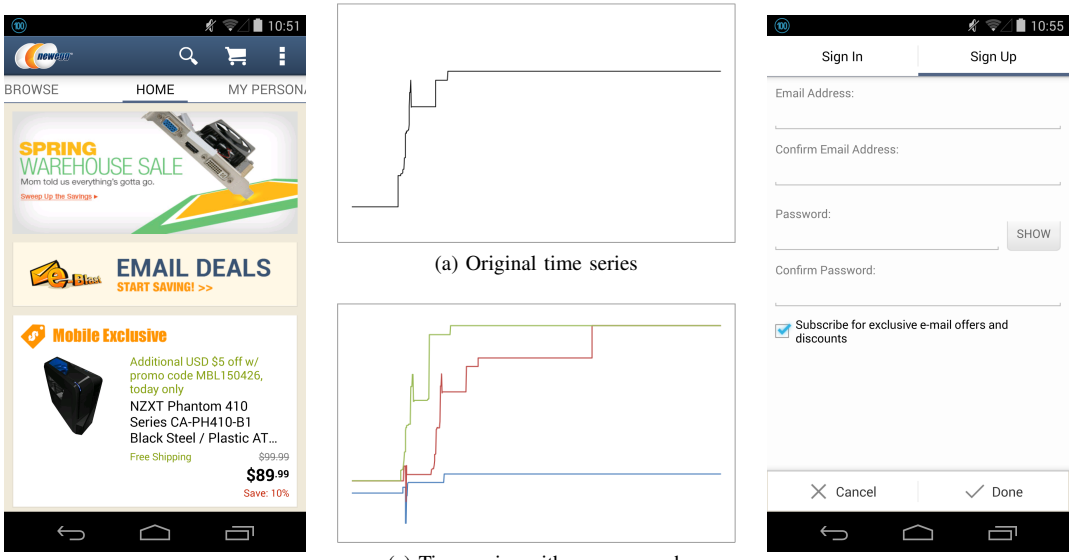
Since restart is such a common and efficient operation on smartphones, and is gracefully tolerated by apps while being disruptive for the OS, our key insight is to use proactive restarts to change the attack surface hence offering a cyber-maneuver capability.

B. Activity Inference Attacks

Activity inference [8] 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. In fact, there is no vulnerability really being exploited, since all the information gathered by the malware M is publicly-available information including `/proc` files, e.g., `/proc/[pid]/statm`.[1]

The fundamental weakness, exploited by such attacks, is that the information exposed through such channels happens to correlate well with B's activity transition behaviors. For instance, when an activity transition occurs in the foreground, the application process allocates 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

[1] Access control on `/proc` is a trade-off between functionality and security. Since many utilities require accessing `/proc`, it is generally undesirable to deny access to it, except in some hardened OSes such as GRSecurity/PaX [3].



(a) Original time series

(b) Main activity

(c) Time series with our approach

(d) Login activity

Fig. 1: Source activity (left), time series of `/proc/[pid]/statm` (center) and destination activity (right).

and therefore cause a new network connection to be created. Through other side channels, such initial behaviors are characterized to distinguish the destination activity.

The Activity inference attack has many consequences, one of which is that 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, hence undermining the attacker's assumption that the side channel is reliable.

III. Example

We now present an example that motivates, as well as illustrates, our approach. Consider the Newegg Mobile app. An attacker might use an Activity Inference attack to try to determine which activity Newegg Mobile is in, and which activity it is transitioning to, so that the attacker can inject its own fake activity to try to phish secrets.

Let us suppose that Newegg Mobile is in the Main activity (Figure 1b) and is preparing to transition to the Login activity (Figure 1d). 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 1a. 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 Login, 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 1c, 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 1c, red and green curves, which represent strategies **S3** and **S4** defined in Section IV). In fact, 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 1c, 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.

IV. Implementation

We now describe our testbed and implementation.

Environment: The smartphone we used for experiments was an LG Nexus 5 running Android version 4.4.4, Linux kernel version 3.4.0, on a four-core ARMv7 CPU@2.2 GHz.

Restart Implementation: In Figure 2 we show our implementation. In Android, applications use the services of the Android Framework (AF) and run on top of the Dalvik virtual machine, which in turn runs on top of a Linux kernel. The AF has a component named Activity Manager (AM) which is in charge of orchestrating app execution, including transition between activities. For simplicity, we only depict one running app, but in practice Android runs multiple apps concurrently. 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), that is, it will transition directly from A to B. In our implementation, the transition follows new pathways (shown in blue color)

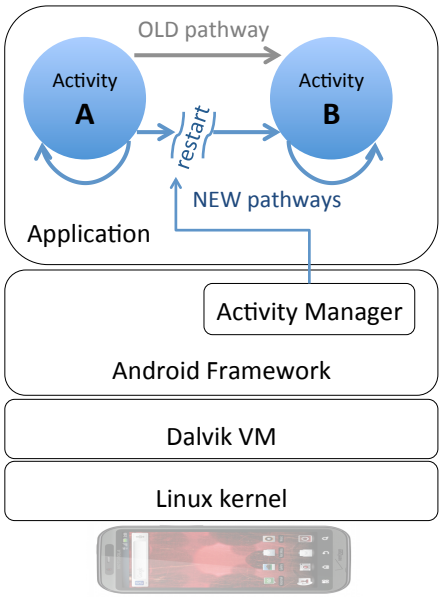


Fig. 2: Overview of our implementation.

where there is an intervening restart, e.g., restart A prior to the transition, or restart B after the transition. Moreover, our approach supports a third new pathway where A is restarted even when no transition is necessary, to confuse the attacker.

We achieve this by using AM services: we use Android's `adb` shell to send messages to the AM, so that activity transitions follow the new pathways. In this paper we use restart level 2 (Section II-A); that is, we stop and restart the activity. Extending the approach to use restart levels 1 or 3 is straightforward.

Restart Strategy: We experimented with four restart strategies, labeled **S1**–**S4**, that govern how the system should proceed when transitioning from activity A to activity B:

- **S1**: The "old" approach, without restart, where we just transition from activity A to activity B.
- **S2**: A restart approach without transition (just restart A).
- **S3**: Our main proposed restart approach: restart A, then transition from A to B.
- **S4**: An alternative restart approach: transition from A to B then restart B.

V. 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.

A. Examined apps

For evaluation we chose 24 activity transitions in 12 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. In Table II we present the apps: name, popularity (number of installs per Google Play), and size. As shown, 9 apps are free and 2 are paid (indicated by the $$ sign). Of the 9 free apps, 5 are third-party apps available on Google Play and 4 are built-in apps that come preinstalled with the phone. The built-in

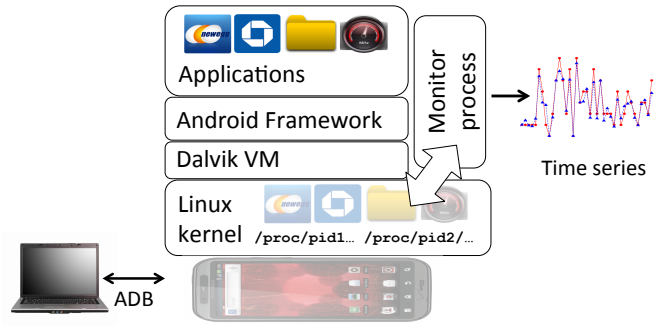


Fig. 3: Overview of our data collection process.

App	Popularity (# installs)	Size (KB)
Chase Mobile	10,000,000+	10,000
Newegg Mobile	1,000,000+	9,900
Browser	(builtin)	2,536
GBC Emulator ($$)	10,000+	367
OI File Manager	5,000,000+	973
Gallery3d	(builtin)	5,122
VideoEditor	(builtin)	5,243
Calendar	(builtin)	1,751
DeskClock	(builtin)	2,311
1MobileMarket	1,000,000+	6,717
Convertor Pro ($$)	10,000+	806
No-frills CPU Control	1,000,000+	1,100

TABLE II: Test apps characteristics.

apps are particularly valuable and need to be protected for two reasons: (1) since they come from a trusted source, the vendor, they have higher privilege than third-party apps and hence exploiting a vulnerability in such an app can inflict significant damage; and (2) preinstalled apps cannot be easily removed by regular users since the phone needs to be "rooted" for the app to be removed [16]. Apps have a range of sizes, from medium (367 KB) to large (10 MB). Four of the third-party apps are very popular, having in excess of 1 million installs. Moreover, two of them — Chase Mobile and Newegg Mobile — are security-critical since they are used for online banking and online shopping; a security attack against them can expose the user's bank account information or credit card numbers.

B. Data collection

The data collection process is shown in Figure 3. The test phone is connected to a laptop via the Android Debugging Bridge (ADB). We triggered restarts using ADB shell commands issued from the laptop. We wrote a monitor process — a native Linux process, rather than VM-based app — to monitor app execution, taking a sample every 8 milliseconds and collecting side-channel information. In particular, the monitor process 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 use time series analysis, as will be explained shortly.

App	Activity Transitions	Permutation Entropy				Transition Time (msec)			
		S1	S2	S3	S4	S1	S2	S3	S4
Chase Mobile	Home → PrivacyOptions	0.332	0.361	**0.683**	0.729	544	248	**1,496**	864
	Home → ContactUs	0.454	0.524	**0.683**	0.349	688	360	**1,432**	1,312
	Home → FindBranch	0.332	0.235	**0.786**	0.332	664	248	**1,640**	784
Newegg Mobile	Main → ShoppingCart	0.256	0.406	**0.696**	0.129	1,088	904	**1,752**	1,088
	Main → WishListItem	0.361	0.332	**0.707**	0.372	1,520	88	**2,120**	1,520
	Main → Login	0.361	0.377	**0.682**	0.361	1,400	928	**2,008**	1,432
	Main → OrderHistory	0.457	0.358	**0.681**	0.364	1,464	112	**2,040**	1,208
	Main → MyPersonalHomeCust	0.358	0.332	**0.722**	0.358	1,360	72	**1,920**	1,328
Browser	BrowserActivity → BrowserPrefsPg	0.332	0.332	**0.595**	0.657	136	24	**648**	672
GBC Emulator ($$)	MainActivity → EmulatorSettings	0.595	0.332	**0.595**	0.595	344	136	**1,040**	352
OI File Manager	FileManager → Preference	0.332	0.352	**0.595**	0.405	136	328	**656**	656
	FileManager → BookmarkList	0.372	0.332	**0.635**	0.657	40	248	**656**	672
Gallery3d	app.Gallery → stngs.GallerySttgs	0.522	0.391	**0.931**	0.489	136	88	**984**	160
VideoEditor	ProjectsActivity → VideoEditorActivity	0.333	0.403	**0.711**	0.333	128	144	**688**	688
Calendar	AllInOneActivity → CalendarSttgs	0.332	0.344	**0.620**	0.332	2,912	256	**3,536**	3,672
	AllInOneActivity → EditEvent	0.355	0.344	**0.724**	0.332	584	192	**2,080**	848
	AllInOneActivity → EventInfo	0.332	0.347	**0.637**	0.687	568	120	**776**	872
DeskClock	DeskClock → SettingsActivity	0.361	0.332	**0.661**	0.689	1,312	320	**1,960**	648
	DeskClock → worldclock.Cities	0.372	0.332	**0.726**	0.701	256	312	**752**	1,936
1MobileMarket	MainActivity → MyAppsInstalledActivity	0.129	0.333	**0.651**	0.801	1,280	1,296	**1,792**	2,064
	MainActivity → SettingsActivity	0.333	0.333	**0.633**	0.333	1,312	192	**560**	2,048
Convertor Pro ($$)	ProConvertActivity → Settings	0.361	0.332	**0.595**	0.457	1,432	424	**2,104**	2,072
No-frills CPU Control	Main → Preferences	0.352	0.332	**0.763**	0.332	136	128	**688**	144
	Main → About	0.332	0.332	**0.332**	0.355	128	8	**664**	128
Average		0.361	0.351	**0.669**	0.465	815	299	**1,416**	1,132
% compared with **S1**			*-3%*	***+85%***	*+29%*		*-63%*	***+74%***	*+39%*

TABLE III: Evaluation results: activity transitions, time series entropy and transition time for each of the **S1**–**S4** strategies.

C. Effectiveness

We now quantify the effectiveness of our approach: we use time series complexity as a measure of effectiveness.

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 a hard time inferring app behavior.

To measure time series complexity, we use the well-known *permutation entropy* (PE) metric [5] normalized so that $0 \leq PE \leq 1$. Here 0 represents no entropy, while 1 represents a random time series. Hence higher PE values are more desirable.

Time series results: Columns 3–7 in Table III show 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**. In the last row, we show the average values across all activities. Note how PE increases from 0.361 on average (**S1**) to 0.669 (**S3**) – an 85% increase, which demonstrates 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 they are used in isolation (though **S4** has a 29% higher PE compared with **S1**).

D. Efficiency

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

The last four columns in Table III show the transition time, from the time the transition was initiated, to when it has completed (including restart time) for each of the four strategies. Compared with **S1**, **S3** increases transition time by 601 msec, i.e., a 74% increase. This exposes the cost-benefit trade-off: we pay the price of increasing transition time for the benefit of increasing attack resilience. Even though our implementation is not optimized, we believe that the added 601 msec are an acceptable overhead, especially when security is of high priority. Strategy **S2** takes less time than **S1** since no transition is involved. Strategy **S4** increases transition time by 317 msec, a 39% increase compared with **S1**.

VI. Related Work

Application Restarts. Application restarts have been used in the past to remedy transient faults. Perkins et al. [12] used a reactive approach, named ClearView, that monitors an application's execution to learn application invariants, detect bugs or attacks, and upon detection automatically construct and apply a patch to heal the application. 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. [7] have proposed "microreboots" (rebooting small components instead of entire applications) as a recovery technique for Internet services. Our own prior work [4] has used online patch construction and application restart to provide self-healing capabilities — apps recovering from certain classes of transient and permanent faults — in Android apps. However, that approach was reactive, rather than proactive, and its goal was fault recovery rather than changing the attack surface.

We are not aware of any work that uses restart as a cyber maneuver.

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. [13] have used "sequence-number-dependent" packet counter side channels to infer TCP sequence number. In Memento [10], the memory footprints were found to correlate with the web page the user is visiting. Zhou et al. [19] found 3 Android/Linux public resources to leak private information about location, disease, etc.. Chen et al. [8] proposed Activity inference attacks that can be applicable to all Android apps.

There are few effective defenses against the types of side-channel attacks. Lately, Zhang et al. have proposed to pause all suspicious background processes to stop them from gathering any data about the foreground app [18]. Such defense could be effective; however, it comes with a functionality cost — many background apps will not be able to function as designed. 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 [6]. Such defense is tailored to attacks similar to Activity inference. In contrast, we believe application restart can be used as a general cyber maneuver against many types of side-channel attacks.

VII. Conclusions and Future Work

We have presented a cyber maneuver for smartphone security: proactively restarting an app to make it harder for attackers to infer app behavior. Since certain side-channel attacks rely on predicting app behavior from OS time series information, to quantify the effectiveness of proactive restart, we use time series predictability. Experiments on 12 popular Android apps, using three proactive restart strategies have revealed that our approach is both effective at reducing side-channel time series predictability (hence increasing attacker's burden) and efficient, imposing an acceptable overhead.

We plan to extend this work in two main directions. First, since in our current approach we control the process from a laptop, we plan to eliminate this need and have the proactive restarts run automatically and completely on the smartphone. Second, we plan to measure the effectiveness of our approach on other kinds of attacks besides Activity Inference.

Acknowledgment

Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] App could allow troops to call in airstrikes, October 2013. http://www.militarytimes.com/article/20131016/NEWS04/310160005/App-could-allow-troops-call-airstrikes.

[2] Army lists top 12 items in fiscal year 2016 budget request, April 2015. http://www.army.mil/article/145946/Army_lists_top_12_items_in_fiscal_year_2016_budget_request/.

[3] Grsecurity, August 2015. https://grsecurity.net/.

[4] M. T. Azim, I. Neamtiu, and L. M. Marvel. Towards self-healing smartphone software via automated patching. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 623–628. ACM, 2014.

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

[6] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the App is That? Deception and Countermeasures in the Android User Interface. In *IEEE Symposium on Security and Privacy*, 2015.

[7] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot: A technique for cheap recovery. pages 31–44.

[8] 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 *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 1037–1052, 2014.

[9] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. Gaur, M. Conti, and M. Rajarajan. Android security: A survey of issues, malware penetration, and defenses. *Communications Surveys Tutorials, IEEE*, 17(2):998–1022, Secondquarter 2015.

[10] S. Jana and V. Shmatikov. Memento: Learning Secrets from Process Footprints. In *IEEE Symposium on Security and Privacy*, pages 143–157, 2012.

[11] P. McDaniel, T. Jaeger, T. F. La Porta, N. Papernot, R. J. Walls, A. Kott, L. Marvel, A. Swami, P. Mohapatra, S. V. Krishnamurthy, and I. Neamtiu. Security and science of agility. In *Proceedings of the First ACM Workshop on Moving Target Defense*, MTD '14, pages 13–19, New York, NY, USA, 2014. ACM.

[12] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[13] Z. Qian, Z. M. Mao, and Y. Xie. Collaborative tcp sequence number inference attack: how to crack sequence number under a second. In *CCS*, pages 593–604, 2012.

[14] S. Sidiroglou, O. Laadan, C. R. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: Automatic software self-healing using rescue points. In *ASPLOS'09*, pages 37–48.

[15] D. Torrieri. Cyber maneuvers and maneuver keys. *Proceedings of the 2014 Military Communications Conference*, 2014.

[16] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 31–40, December 2012.

[17] K. Zhang and X. Wang. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In *USENIX Security Symposium*, pages 17–32, 2009.

[18] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave Me Alone: App-level Protection Against Runtime Information Gathering on Android. In *IEEE Symposium on Security and Privacy*, 2015.

[19] 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 *CCS*, pages 1017–1028, 2013.