ProcHarvester: Fully Automated Analysis of Procfs Side-Channel Leaks on Android

Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard
Graz University of Technology

ABSTRACT

The procfs has been identified as a viable source of side-channel information leaks on mobile devices. Starting with Android M (Android 6), access to the procfs has been continuously restricted in order to cope with these attacks. Yet, more recent papers demonstrated that even if access to process-specific information is restricted within the procfs, global statistics can still be exploited. However, with state-of-the-art techniques, the search for procfs information leaks requires a significant amount of manual work. This makes an exhaustive analysis of existing and newly introduced procfs resources in terms of information leaks impractical.

We introduce ProcHarvester, a systematic and fully automated technique to assess procfs information leaks. ProcHarvester automatically triggers events of interest and later on applies machine learning techniques to identify procfs information leaks. We demonstrate the power of ProcHarvester by identifying information leaks to infer app starts from a set of 100 apps with an accuracy of 96% on Android N (Android 7). Thereby, we outperform the most accurate app inference attack by about 10 percentage points. We also demonstrate the ease of applicability of ProcHarvester by showing how to profile other events such as website launches as well as keyboard gestures, and we identify the first procfs side channels on Android O (Android 8). ProcHarvester advances investigations of procfs information leaks to the next level and will hopefully help to reduce the attack surface of side-channel attacks.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**;

KEYWORDS

Android; automatic analysis; procfs; side-channel analysis

ACM Reference Format:
Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. 2018. ProcHarvester: Fully Automated Analysis of Procfs Side-Channel Leaks on Android. In *ASIA CCS '18: 2018 ACM Asia Conference on Computer and Communications Security, June 4–8, 2018, Incheon, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3196494.3196510

1 INTRODUCTION

Side-channel attacks exploit information leaks of computing platforms in order to learn sensitive information about users as well as their computing devices and the processed data. Especially side-channel attacks on mobile devices have gained particular attention and manifold attack possibilities have been suggested to extract secret keys from cryptographic implementations, to bypass security mechanisms, to infer keyboard input and user behavior, etc. Existing attacks range from, for example, sensor-based keyloggers [5, 6, 16, 22, 25], via micro-architectural attacks [15, 26, 29, 30, 37], to attacks exploiting information leaks from the virtual file system mounted under `/proc/` (procfs) [14, 18, 24, 34]. Especially the procfs has been identified as an apparently unlimited source of information leaks. For example, procfs information has been used to infer inter-keystroke timings [35], keyboard input [24], unlock patterns [12], user identities and diseases [38], a user's location [18], visited websites [14, 27], and user interfaces [7, 12, 34].

A fundamental weakness of the procfs is the availability of process-specific information, e.g., in `/proc/uid_stat/<uid>/*`, and `/proc/<pid>/*`. As the majority of procfs side-channel attacks exploit per-process information, access to procfs resources has been continuously restricted since Android M (Android 6). Although these restrictions mitigate attacks that exploit process-specific information (`/proc/<pid>/*`), newer attacks exploit global procfs information that is still available. For instance, Simon et al. [24] inferred swipe input on soft-keyboards by exploiting interrupt information (`/proc/interrupts`) and the number of context switches (`/proc/stat`). Diao et al. [12] inferred unlock patterns and running applications (apps) via interrupt statistics. As of Android O (Android 8) access to global interrupt statistics has also been removed.

This trend illustrates the arms race between OS designers aiming to reduce the attack surface and attackers aiming to find new information leaks. Furthermore, as claimed in many of these papers, the identified information leaks represent just the tip of the iceberg and more information leaks are yet to be discovered. Therefore, we aim for a systematic analysis of procfs information leaks. We introduce ProcHarvester,[1] a tool that automatically profiles procfs information for events of interest. More specifically, ProcHarvester finds correlations between events of interest and procfs information.

We demonstrate the applicability of ProcHarvester by automatically identifying new as well as existing information leaks. As a proof of concept, we analyze app inference attacks. ProcHarvester automatically launches applications of interest and simultaneously samples procfs resources. In this setting, ProcHarvester outputs a list of procfs files and properties that can be exploited in side-channel attacks to infer app launches. We compare our results to existing app inference attacks and show that the side channels found by ProcHarvester outperform existing attacks. The identified information leaks allow to infer app starts from a set of 100 apps with an accuracy of 96% on Android 7, which increases the most

ASIA CCS '18, June 4–8, 2018, Incheon, Republic of Korea

[1] We responsibly disclosed our findings to Google. The ProcHarvester framework is available at: https://github.com/IAIK/ProcHarvester.

accurate attack so far [12] by about 10 percentage points. Besides, we demonstrate how ProcHarvester can be used to systematically search for information leaks that allow to infer visited websites, as well as keyboard gestures on soft-keyboards. These examples are by no means exhaustive, but illustrate the power of ProcHarvester.

Contributions. Our contributions are as follows:

(1) We introduce ProcHarvester, a fully automated technique to find procfs leaks, even on already hardened Android systems, and identify exploitable side-channel leaks on Android N (Android 7) as well as the new Android O (Android 8).
(2) We demonstrate the generic methodology of ProcHarvester by automatically detecting procfs information leaks that allow to infer sensitive events such as app starts, website launches, and soft-keyboard gestures.
(3) We reveal new attack surfaces within the procfs that allow to precisely infer application launches and thereby outperform the most accurate state-of-the-art attacks.

Outline. In Section 2, we discuss the procfs and related work. In Section 3, we discuss the principle of automatically profiling the procfs with ProcHarvester. In Section 4, we demonstrate how to profile the procfs for information leaks that allow to infer app starts and we evaluate the identified procfs leaks. In Section 5 and Section 6, we show how to use ProcHarvester to profile website launches and keyboard gestures. In Section 7, we discuss countermeasures, how ProcHarvester helps to reduce the attack surface of procfs side-channel attacks as well as limitations and the performance of our framework. Finally, we conclude in Section 8.

2 BACKGROUND AND RELATED WORK

2.1 The Linux procfs

The process information file system (procfs) is a virtual file system mounted under /proc/ on Linux-based operating systems, including Android. As the name suggests, it provides information about processes running on the system. For example, information about shared memory is available via /proc/<pid>/statm for a given process ID (<pid>), and network traffic statistics are available via /proc/uid_stat/<uid>/[tcp_rcv|tcp_snd] for a given user ID (<uid>). Since Android apps are assigned a user ID (uid) during the installation, and a process ID (pid) identifies an executed process, these resources provide per-application information. Besides per-application information, the procfs also provides global information which is considered innocuous, e.g., statistical information about processed interrupts on the system via /proc/interrupts. In addition to the procfs, Linux-based operating systems provide information about the hardware and the device via the sysfs (/sys/).

procfs Restrictions. Since Android 4.3, SELinux [4] further restricts apps by means of mandatory access control (MAC), which allows more fine-grained access control policies than discretionary access control (DAC). In general, third-party apps are associated with the label untrusted_app, and system apps are associated with the label system_app. Since Android M (Android 6),[2] apps running as untrusted_app have been restricted to access only /proc/ entries of other apps running with label untrusted_app. Starting with Android N (Android 7)[3] the procfs is mounted with hidepid=2, *i.e.*, processes cannot access /proc/<pid>/* for a pid other than their own. In Android O (Android 8) the procfs is restricted even further, e.g., /proc/interrupts is not available anymore.

2.2 Related Work

Side-channel attacks on mobile devices exploit shared resources, e.g., sensors [5, 6, 19, 22, 25, 33] or microarchitectural components [15, 26, 29, 30, 37], to infer sensitive information such as keystrokes and keyboard input as well as cryptographic keys. Other well-known attacks include the exploitation of sensor information to infer a user's location and traveling patterns [13, 21], to fingerprint devices [9–11, 39], and to eavesdrop conversations [17]. As the set of information leaks on mobile devices is quite diverse, we refer to [28] for a survey of side-channel attacks on mobile devices.

In this work, we focus on the exploitation of procfs interfaces. We discuss side-channel attacks that exploit procfs interfaces below.

Keylogging and Unlock Pattern Attacks. Zhang and Wang [35] published one of the first papers exploiting the procfs. They observed that the stack pointer (ESP) in /proc/<pid>/stat allows to monitor inter-keystroke timings. Simon et al. [24] inferred swipe input on soft-keyboards running on Android $\geq$ 4.4 by exploiting global interrupt statistics available via /proc/interrupts. Furthermore, Diao et al. [12] presented an attack to infer unlock patterns by also exploiting touchscreen interrupt statistics on Android 5.1.1.

Inference of User Information. Jana and Shmatikov [14] exploited the memory footprint (/proc/<pid>/statm) and the number of context switches (/proc/<pid>/status) of the browser to infer visited websites. Zhou et al. [38] inferred diseases by monitoring traffic statistics (/proc/uid_stat/<uid>/[tcp_rcv|tcp_snd]) of applications, and Spreitzer et al. [27] inferred visited websites based on the traffic statistics of the browser.

Michalevsky et al. [18] observed that the power consumption (/sys/class/power_supply/battery/*) correlates with the cellular signal strength, which allows to infer a user's location.

Application and Activity Inference. Chen et al. [7] proposed a user interface (UI) inference attack that exploits the size of the shared memory of specific apps (/proc/<pid>/statm). Since shared memory is used for the communication between an app and the process that updates the frame buffer, the size of the shared memory indicates activity transitions. Subsequently, they rely on the CPU utilization time (/proc/<pid>/stat), the size of transmitted network packets (/proc/uid_stat/<uid>/tcp_snd), and destination IP addresses (proc/net/tcp6) to infer the activity. They relied on Android 4.2 and considered 7 different apps. Similarly, Yan et al. [34] inferred apps and activities by exploiting the power consumption (/sys/class/power_supply/battery/*). They were able to distinguish 3 different apps as well as 3 activities within the Amazon app on Android 4.4. Recently, Diao et al. [12] exploited the interrupt counter of the display sub-system (MDSS) (/proc/interrupts) to infer running apps. They collected training data for 100 apps and randomly selected 10 apps for their attack. For these 10 apps they report a success rate of 87% on Android 5.1.

[2] android-review.googlesource.com/#/c/105337/.

[3] android-review.googlesource.com/#/c/181345/.

Table 1: Devices used for the practical experiments.

Device	Operating system
One Plus 3T	Android 7.1.1 (LineageOS)
Sony Xperia Z5	Android 7.0 (Stock ROM)
Emulator (Nexus 5X)	Android 8.0 (Developer preview)

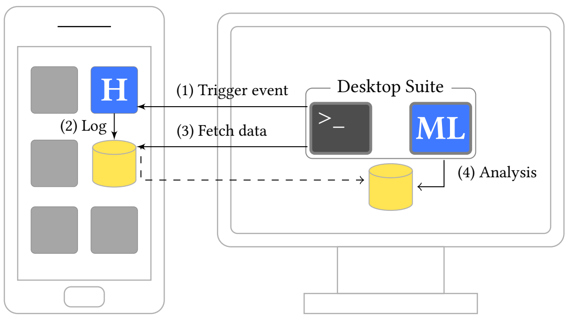

Figure 1: Basic design and work flow of ProcHarvester.

As discussed in Section 2.1, access to `/proc/<pid>/` and `/sys/` has been restricted in Android 7.[4] Therefore, more recent attacks [12, 24] exploit global procfs information, e.g., `/proc/interrupts`, but Android 8 also restricts access to global interrupt information.

2.3 Test Devices

For the practical experiments, we rely on the Android devices as shown in Table 1. We explicitly focus on Android 7 as it already restricts many of the previously exploited information leaks and investigations on Android 7 are quite scarce. Furthermore, we provide first insights about side-channel leaks on the new Android 8.

3 THE PROCHARVESTER FRAMEWORK

ProcHarvester enables a systematic analysis of information leaks by automatically profiling procfs behavior for events of interest. Considering the attacks discussed above, ProcHarvester triggers specific events—e.g., app starts, website launches, keystrokes, etc.—and scans the procfs for information leaks that allow to infer the corresponding events later on.

Template Attacks. Our approach is based on template attacks, where templates for events of interest are modeled. Later, one observes the leaking information and infers the events by means of these templates. The appealing benefit of this methodology is that information leaks can be analyzed without background knowledge of the underlying effects. Thus, this approach is perfectly suitable for an automatic analysis of procfs leaks.

Based on template attacks, ProcHarvester finds correlations between triggered events and procfs information, which can be exploited for side-channel attacks. Figure 1 depicts the design of ProcHarvester consisting of an Android app (H) and a Desktop Suite. The Android app systematically logs procfs information. The Desktop Suite consists of a tool to control the Android app as well as the device via the Android Debug Bridge (ADB) [2], e.g., to trigger events of interest, and a machine learning framework (ML) to analyze the gathered data in terms of information leaks.

[4] code.google.com/p/android/issues/detail?id=208085.

ProcHarvester works in four phases: exploration phase, profiling phase, analysis phase, and attack phase. The work flow of ProcHarvester is as follows.

(1) Trigger Event: The Desktop Suite triggers events via the ADB connection. Besides triggering events via ADB capabilities, the framework can trigger events by other means as well, e.g., via the *MonkeyRunner* [3], programmatically via the Android app (H) itself, and events can also be triggered by a human being.
(2) Log: The Android app (H) identifies potential information leaks from procfs resources in the *exploration phase*. Irrespective of the actual approach to trigger events, the Android application continuously monitors and logs the procfs resources in the subsequent *profiling phase*, *i.e.*, while events are triggered.
(3) Fetch Data: After the events have been triggered, the log files are fetched to the Desktop Suite for the subsequent analysis.
(4) Analysis: In the *analysis phase*, the gathered time series are analyzed for possible correlations in order to identify information leaks that allow to infer the triggered events. The output is a list of resources that can be exploited in side-channel attacks, *i.e.*, in the *attack phase*, to infer the triggered events.

3.1 ProcHarvester Android App

ProcHarvester runs as an *IntentService* in the background and samples the procfs. Experiments on our test devices revealed a sampling frequency of 200 Hz for logging about 20 procfs resources at the same time. For a systematic and possibly exhaustive analysis of procfs leaks, resources can be logged during subsequent executions.

Triggering Events. Events can be triggered within the ProcHarvester Android app directly—either programmatically or by a human being—or via the ADB shell. Naturally, when programmatically triggering events within the Android app, dedicated permissions might be required during the analysis phase, but no permissions are required for the exploitation of the identified procfs leaks.

The Android app implements the `CommandReceiveActivity` to handle various *Intents*, which are used to execute commands via the ADB shell. More specifically, commands and optional arguments can be passed to this activity via *Extra Data* supported by the *Intent*.

Identification of Target Resources. As we are interested in publicly readable files within the procfs, we identify such files based on file permissions. Files that seem to be publicly readable due to the DAC mechanism, but are further restricted due to the MAC mechanism (cf. SELinux since Android 4.3) are filtered out in the profiling phase, as they cannot be accessed by zero-permission apps.

Exploration. Before the profiling starts, the *exploration* phase automatically identifies possible information leaks in the targeted procfs files. The app parses numerical values in the corresponding files and keeps track of the line indices and column indices. During this exploration phase we also trigger events of interest to induce possible information leaks. A resource is considered in the subsequent profiling phase if it changes with a sufficiently high frequency, depending on a configurable threshold. The threshold represents an optimization parameter and restricts the search space to information leaks that are non-static. For an exploration phase of 6.5 seconds, we fixed the threshold to 10, *i.e.*, we focus on resources that change with a frequency of more than $\frac{10}{6.5} \approx 1.5$ Hz. Hence, we follow a more conservative approach than existing (manual) attacks,

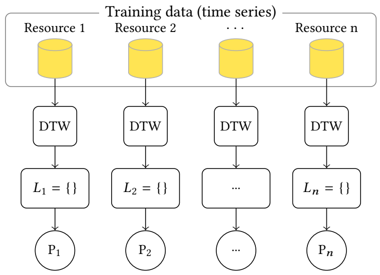

Figure 2: Strategy in single-resource mode.

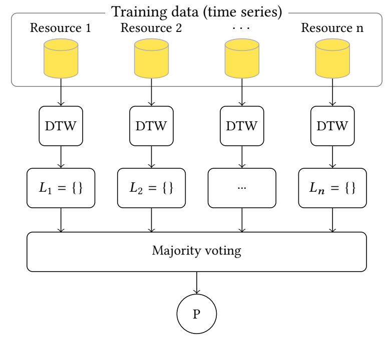

Figure 3: Strategy in multi-resource mode.

which consider sampling frequencies of 10–1000 Hz [12, 18, 38], *i.e.*, resources that change more frequently. Nevertheless, the threshold could also be set to 1, resulting in a more expensive profiling phase.
Profiling. After the exploration phase, the *profiling* phase starts. In this phase, time series of previously identified candidate side channels (based on the line indices and column indices) are logged to separate files while events of interest are triggered simultaneously.

3.2 ProcHarvester Desktop Suite

The Desktop Suite consists of two parts: A tool to send ADB commands and an analysis tool. This allows to automatically trigger events of interest on the device—in case the events are not triggered directly on the smartphone—and also to transfer the gathered information to the Desktop Suite for the subsequent analysis.
Triggering Events and Sending ADB Commands. Currently, ProcHarvester supports triggering app launches, website launches, and tap and swipe actions. However, ProcHarvester can easily be extended to be applicable to other events as well. Besides triggering events on the device, commands allow to start and stop the logging service, and to communicate the corresponding label of the event to the ProcHarvester Android app in order to label the gathered data for the analysis. All events of interest are triggered in a randomized order to simulate a more realistic usage scenario.
Machine Learning Methodology. The Desktop Suite also analyzes the gathered information for information leaks. Therefore, it relies on the machine learning framework *scikit-learn* [23]. In a preprocessing step, we normalize time series by subtracting the mean. Afterwards, we rely on dynamic time warping (DTW)[5] to identify similarities between gathered time series of procfs resources for the triggered events. Given two time series $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_m)$ of (possibly) different lengths, DTW compares these two time series by finding a warping path with minimal distance. For classification purposes, a time series X is matched against other time series Y_i to find a class i, such that $i = \operatorname{argmin} \mathrm{DTW}(X, Y_i)$. We implicitly assume that two time series originate from the same target label (class) if they yield a low distance to each other based on DTW. The appealing benefit of DTW is that possibly misaligned time series can be compared (cf. [20]) without background knowledge about information leaks and without human interaction.

Although supervised classifiers based on features manually identified by an expert would probably yield even better results than our approach, we focus on a fully automated technique that does not require any human interaction. Hence, we also investigated the use of automatic feature extraction, by using *tsfresh* [8], in order to train common supervised learning algorithms such as KNN classifiers and multi-class SVMs. However, we found the accuracy of information leaks identified through DTW to be significantly higher than with supervised learning algorithms based on automatic feature extraction. This shows that common supervised learning algorithms cannot easily be adapted for a fully automated approach.
Analysis Modes. The analysis tool can be used in two modes, namely single-resource mode and multi-resource mode.

(1) In *single-resource mode*, we evaluate the accuracy of inferring events based on a single resource at a time. Figure 2 depicts the basic principle. The following k-nearest-neighbor approach is used to classify time series. We determine the top k labels (L_i) of the k events in the training data with the smallest DTW distances to the time series to be classified, where the training data consists of multiple time series for each event of interest and procfs resource. Based on this list of k labels, the majority of the reported labels is used to predict the most likely one (P_i).
(2) In *multi-resource mode*, multiple resources are evaluated simultaneously and the results of all resources are combined by a majority voting to evaluate the overall performance of a specific combination of information leaks. Figure 3 depicts this strategy. The top k labels of each single resource are collected in a list (L_i) and the majority of the reported labels for all these resources then determines (predicts) the event. Without prior knowledge on the exploited information and considering possibly noisy side channels, majority voting allows us to combine multiple resources and to determine the most likely event. Hence, the multi-resource mode automatically evaluates possible attacks that exploit multiple resources at the same time.

In an actual attack one might extract more specialized features from the identified information leaks, which might lead to even higher accuracies. We, however, focus on a general approach to identify information leaks automatically and do not rely on specialized features in order to launch fully-fledged attacks. Nevertheless, the generic approach of DTW allows us to automatically identify information leaks and to launch sophisticated generic side-channel attacks based on the identified information leaks.
Summary of Methodology. An important advantage of ProcHarvester is that a thorough understanding of the actual information leak is not necessary to detect it. Our proposed methodology identifies information leaks in a fully automated fashion as we establish

[5] https://github.com/honeyext/cdtw.

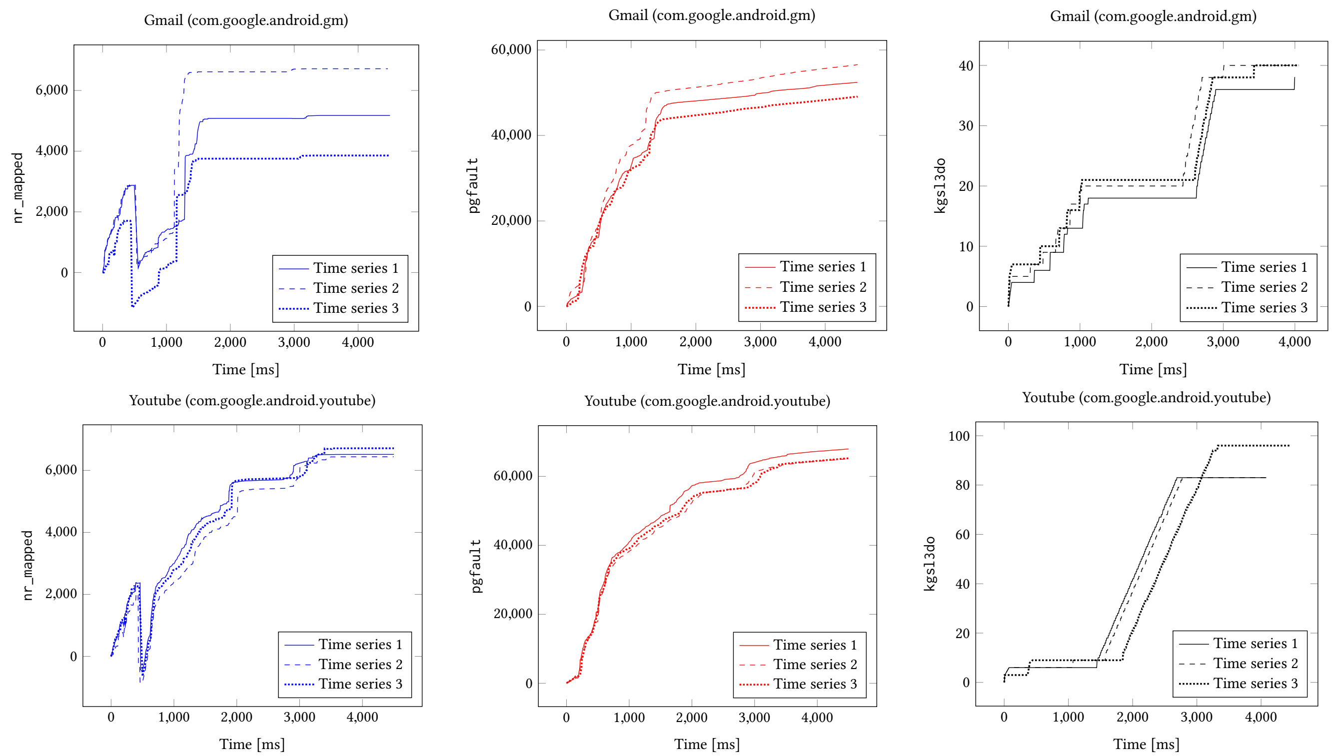

Figure 4: Information leaks (`nr_mapped`, `pgfault`, and `kgsl3do`) for Gmail and Youtube.

templates—for events of interest—which are then used to identify information leaks. Due to this automatic approach, ProcHarvester allows to quickly analyze possible attacks on different Android versions, e.g., AOSP as well as vendor-specific ones. In this work, we focus on events of interest that are already known, *i.e.*, already known attacks, to demonstrate the power of ProcHarvester.

4 APPLICATION INFERENCE

To demonstrate the applicability of ProcHarvester, we analyze the procfs during app starts. The learned information allows us to perform application inference attacks from an unprivileged app by monitoring the automatically identified procfs resources.

Problem Description. Currently executed apps should be kept secret as this information enables targeted attacks, e.g., phishing attacks [7] that steal login credentials. Thus, Android prevents third-party apps from learning currently executed applications. Up to Android L (Android 5), the `GET_TASKS` permission allowed to obtain running apps via `ActivityManager.getRunningTasks()` and `ActivityManager.getRecentTasks()`. In Android L (Android 5), `GET_TASKS` has been replaced with the permission `REAL_GET_TASKS`, which is not granted to non-system apps anymore.

4.1 Profiling

We instructed ProcHarvester to profile app starts, as shown in Listing 1. Although ProcHarvester uses internal methods to handle ADB commands as well as to start and stop the logging app, we provide them here for the sake of clarity and to illustrate the basic communication between the Desktop Suite and the Android app.

Listing 1: Profiling app starts with ProcHarvester.

```
# Repeat for all apps (<package>)
adb shell am start \
  -n com.harvester.CommandReceiveActivity \
  --es CMD TRIGGER_EVENT --es ARG <package>
adb shell monkey -p <package> \
  -c android.intent.category.LAUNCHER 1
sleep 4.5 # logging stops after 4 seconds
adb shell am force-stop <package>
```

4.2 Analysis and Evaluation on Android 7

4.2.1 Information Leaks. In the analysis phase, ProcHarvester identified several procfs resources that allow to infer app starts. The evaluation presented in this section is based on experiments with the One Plus 3T. Experiments on the Xperia Z5 revealed almost identical results and, hence, have been omitted.

Figure 4 illustrates three identified information leaks for Gmail, and Youtube, respectively. We observe that multiple starts of the same app lead to similar time series and that time series for different apps can be distinguished. These plots also illustrate that relying on DTW to identify correlations yields reliable results regarding information leaks, since DTW aims to identify similarities between sequences that vary in time or speed (cf. [20]). Therefore, these time series serve as templates for the subsequent evaluation.

Table 2 provides an excerpt of procfs leaks that allow to infer app starts on Android 7. The accuracy has been evaluated for the 100

Table 2: Excerpt of identified information leaks for app inference on Android 7. Accuracy evaluated for 100 apps.

procfs file	Property	Accuracy
/proc/vmstat	nr_mapped	82.2%
/proc/vmstat	pgfault	73.3%
/proc/interrupts	kgsl3do	71.5%
/proc/vmstat	nr_anon_pages	71.3%
/proc/interrupts	arch_timer	70.1%
/proc/interrupts	MDSS	67.6%
/proc/interrupts	Rescheduling interrupts	62.9%
/proc/vmstat	nr_dirty_threshold	62.2%
/proc/vmstat	nr_shmem	58.9%
/proc/vmstat	nr_free_pages	49.1%
/proc/interrupts	Single function call interrupts	48.3%
/proc/interrupts	dwc3	47.2%
/proc/net/sockstat	Sockets used	74.1%
/proc/net/dev	wlan0: Receive bytes	73.8%
/proc/net/dev	wlan0: Transmit bytes	68.4%
/proc/sys/fs/inode-state	nr_inodes (column 0)	65.0%
/proc/meminfo	VmallocUsed	55.9%
/proc/sys/fs/dentry-	nr_dentry (column 0)	54.1%
/proc/pagetypeinfo	zone DMA, type Unmovable	39.7%
/proc/schedstat	cpux (column 8: Time spent waiting to run)	37.8%

apps listed in Appendix A. For ProcHarvester the exact meaning of these properties does not matter. The idea is to report properties for which a correlation between time series could be observed since these properties allow to identify the corresponding event later on. Nevertheless, we indicate named properties within the procfs files as property_name and in case of unnamed properties we provide the column number (starting at 0) within the procfs file.

As there are multiple columns in /proc/interrupts (one for each CPU) and we do not know on what CPU the targeted event is executed, we simply sum all interrupt counters from the individual CPUs. The information leaks resulting from the Mobile Display Sub-System (MDSS) have already been exploited by Diao et al. [12] to perform app inference attacks. However, we still report it here since ProcHarvester automatically identified MDSS as an information leak. To the best of our knowledge, the other information leaks identified by ProcHarvester have not been reported so far.

4.2.2 Adversary Model and Evaluation. Based on the observed information leaks, we evaluate the performance of fingerprinting app starts. Therefore, we assume an adversary model where a user installed a malicious app on her device. As the app does not require any permission, the user will not notice anything suspicious during the installation. We rely on an analysis phase where the adversary gathers the identified procfs resources for applications of interest to establish the application fingerprint database, *i.e.*, the templates for specific apps of interest. This analysis phase, *i.e.*, the gathering of templates, can be done on the targeted device or on a device controlled by the adversary. During the attack phase, the malicious application monitors the previously identified information leaks and exploits this information to infer application launches. For our evaluation, the profiling phase and the attack phase have been performed on the same device, as also done in the studies we compare our results to [12, 34].

Evaluation. For the subsequent evaluation we establish a database of fingerprints for the 100 apps listed in Appendix A. We collected 10 samples, *i.e.*, 10 time series for the procfs leaks in the upper part of Table 2, per app and considered the following four scenarios.

Table 3: App inference attacks on Android 7 based on identified information leaks for application cold starts.

Attack	# Apps	Accuracy
App cold starts	100	96%
App resumes	20	86%
Mixed (cold starts and app resumes)	20	90%
Manual cold starts (by human being)	20	98%

App cold starts: By combining the identified information leaks by means of majority voting (in the multi-resource mode), we achieve an average classification rate of 96% based on 8-fold cross validation for all 100 apps. We significantly outperform the most accurate attack by Diao et al. [12], who report an accuracy of 87% for 10 randomly chosen apps out of 100 apps. The detailed results for app cold starts can be found in Appendix A.

App resumes: We also evaluated the accuracy of inferring app resumes with the identified information leaks for app cold starts. Although a dedicated profiling phase will most likely identify further information leaks that allow to infer app resumes more accurately, we achieve an average classification rate of 86% for 20 applications, selected randomly out of the 100 applications. This shows that even if the attacker has only templates for app cold starts, app resumes can still be monitored with a high accuracy. The detailed classification results for application resumes can be found in Appendix A.

Mixed: As seen in the previous two cases, we are able to identify app cold starts as well as app resumes by relying on the templates for app cold starts. We evaluated the combination of these two cases, *i.e.*, app cold starts and app resumes, by randomly selecting 20 applications out of the 100 applications and achieved an average classification rate of 90% based on k-fold cross validation. The detailed classification results for app cold starts and app resumes can be found in Appendix A.

Manual cold starts: Since we gathered the training data by triggering the app starts automatically via the ADB shell, we also verified the identified side channels manually. Therefore, we launched 20 apps, each 10 times, by manually tapping the application icon with a finger while monitoring the identified resources in the background. During these manual application starts, the dwc3 interrupt (in /proc/interrupts) did not leak any information on manual app starts. Instead, we found that the dwc3 interrupt is caused by the USB interface, representing a new side channel that allows to spy on USB connections.
The remaining information leaks presented in Table 2 were also exploitable during manual app starts. This indicates that most of the identified information leaks do not differ between programmatically triggered events and manually (by a human being) triggered events, which strengthens the approach of automatically identifying information leaks. The detailed results for manual application cold starts can be found in Appendix A.

Table 3 summarizes our investigations. All accuracies have been averaged by means of k-fold cross validation.

Table 4: Excerpt of identified information leaks for app inference on Android 8. Accuracy evaluated for 20 apps.

procfs file	Property	Accuracy
/proc/net/sockstat	sockets: used	86.3%
/proc/net/xt_qtaguid/ iface_stat_all	eth0: tx_packets (column 9)	77.2%
/proc/net/xt_quota/eth0	eth0: interface quota	76.9%
/proc/net/protocols	UNIX: sockets	76.3%
/proc/net/xt_qtaguid/ iface_stat_fmt	eth0: total_skb_tx_packets	76.3%
/proc/meminfo	AnonPages	76.3%
/proc/meminfo	Active(anon)	75.9%
/proc/meminfo	MemFree	70.9%
/proc/meminfo	Mapped	62.5%
/proc/meminfo	Shmem	55.0%

Table 5: Comparison of app inference attacks. ✓ and ✗ indicate whether the attack works on a specific Android version.

Work	procfs information	# Apps	Accuracy	Android 7	Android 8
Yan et al. [34]	/sys/.../battery	3	100%	✗	✗
Diao et al. [12]	/proc/interrupts	10/100	87%	✓	✗
Ours	/proc/interrupts, /proc/vmstat (Table 2)	100/100	96%	✓	✗
Ours	/proc/meminfo (Table 4)	20/100	87%	✓	✓

4.3 Analysis and Evaluation on Android 8

Similar to the evaluation on Android 7, ProcHarvester identified information leaks that allow to infer app starts on Android 8. The profiling and evaluation are performed exactly as on Android 7.

Table 4 provides an excerpt of the information leaks and the corresponding accuracies evaluated for app starts on Android 8. We observe that on Android 8 /proc/vmstat is not available anymore. However, most of the information that has been published in /proc/vmstat is still available in /proc/meminfo. Thus, the information leaks have not been closed, but instead the information is available at a different location within the procfs. Since the experiments on Android 8 have been carried out with an emulator, some of the procfs leaks are related to the Ethernet network interface (eth0) instead of the Wi-Fi network interface (wlan0). Nevertheless, running ProcHarvester on a real device will yield similar results.

Combining the information leaks in the lower part of Table 4 yields an average classification rate of 87% based on k-fold cross validation. Appendix B depicts the detailed results for all 20 apps.

4.4 Comparison of Attacks

Table 5 compares our results to related work. Access to /sys/ has been restricted in Android 7 and, hence, the information leak exploited by Yan et al. [34] does not work anymore. Compared to the previously most accurate attack by Diao et al. [12], ProcHarvester automatically identified information leaks that allow us to significantly outperform their attack. Besides, Diao et al. [12] report an accuracy of 87% for 10 randomly chosen apps out of 100 apps, whereas we are able to infer 96% of all 100 apps. In addition, the attack by Diao et al. [12] does not work on Android 8 since /proc/interrupts is not available anymore.

We stress that the main intention of this work is to demonstrate the strength of ProcHarvester in identifying information leaks automatically. Hence, we also do not focus on a stealthy attack considering, e.g., the battery consumption of the Android app. Nevertheless, with our fully automated attacks, we outperform the most accurate attack to date on Android 7 and we present the first procfs-based side-channel attack on Android 8.

5 WEBSITE INFERENCE

We also instructed ProcHarvester to investigate procfs leaks that can be exploited for website fingerprinting attacks [14, 27].

Problem Description. A user's browsing behavior reveals sensitive information such as sexual orientation, diseases, etc. Therefore, up to Android M (Android 6) it has been protected by means of the READ_HISTORY_BOOKMARKS permission, and starting with Android M access has been removed entirely [1].

5.1 Profiling

In order to investigate information leaks in the procfs that allow to infer visited websites, we instructed ProcHarvester to profile website launches via the *Chrome* browser, as shown in Listing 2.

Listing 2: Profiling websites with ProcHarvester.

```
# Repeat for all websites (<URL>)
adb shell am start \
  -n com.harvester.CommandReceiveActivity \
  --es CMD TRIGGER_EVENT --es ARG <URL>
adb shell am start \
  -a "android.intent.action.VIEW" -d <URL>
sleep 4.5 # logging stops after 4 seconds
# Kill the browser
adb shell am force-stop com.android.chrome
```

5.2 Analysis and Evaluation on Android 7

5.2.1 Information Leaks. ProcHarvester identified several resources in the procfs that allow to fingerprint websites and, thus, to infer a user's browsing behavior. Again, the evaluation is based on experiments with the One Plus 3T. Experiments on the Xperia Z5 revealed almost identical results and, hence, have been omitted.

Figure 5 depicts three identified procfs leaks for facebook.com, and wikipedia.org, respectively. Again, we observe that multiple visits to the same website lead to similar time series and that time series for different websites can be distinguished. Since we use DTW as a similarity measure, misalignments are entirely negligible. Specifically, the time series for wikipedia.org have visually observable time offsets, but DTW correctly detects the similarity.

Table 6 provides an excerpt of the identified information leaks that allow to fingerprint websites. Most information leaks are related to statistics collected about the number of packets received and transmitted as well as the number of bytes received and transmitted. Furthermore, the number of pages used for shared memory also leaks information about visited websites. Nevertheless, we do not aim to interpret the automatically identified information leaks and ProcHarvester reports information leaks irrespective of the actual information that leaks and without background knowledge. Hence, also redundant procfs resources, such as IpExt: InOctets and wlan0: Receive bytes, have been identified. We evaluated the detection accuracy for the top 20 websites according to alexa.com.

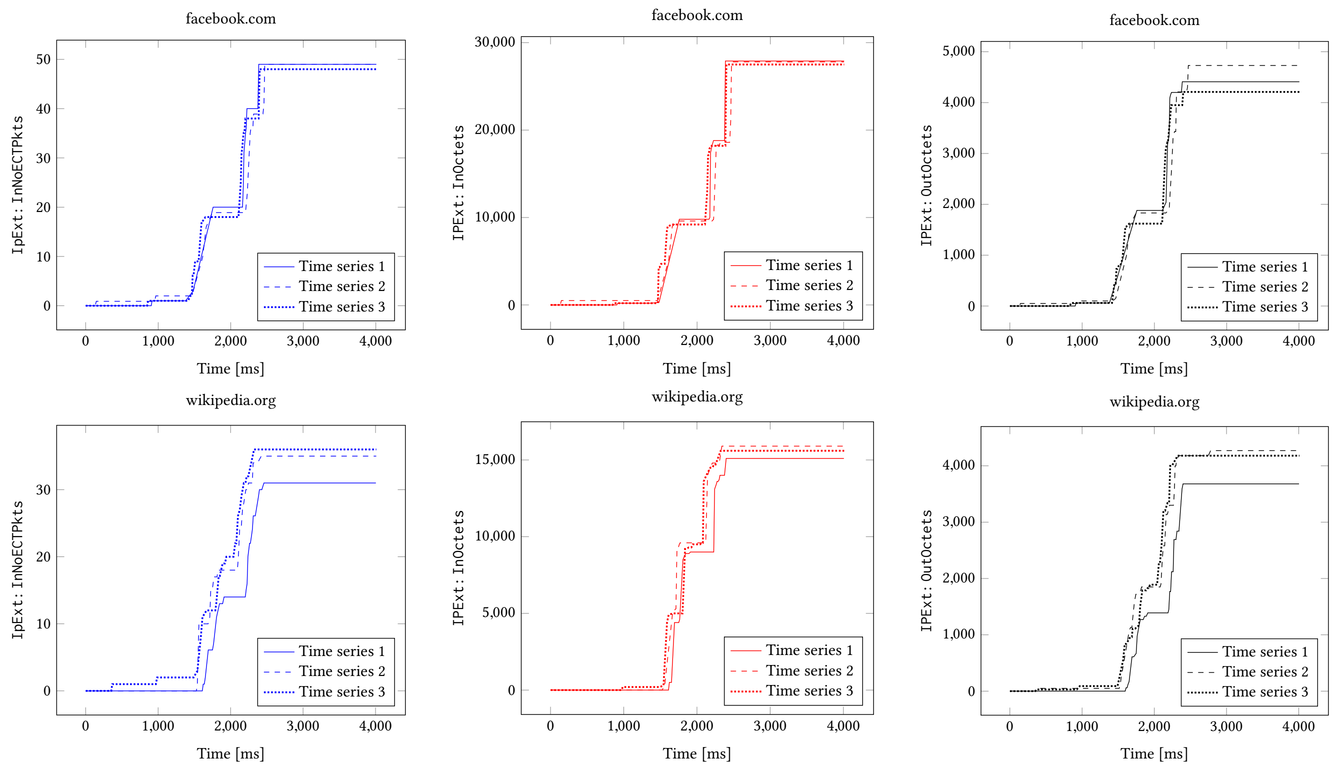

Figure 5: Information leaks (`IpExt: InNoECTPkts`, `IPExt: InOctets`, and `IPExt: OutOctets`) for facebook.com and wikipedia.org.

Table 6: Excerpt of identified information leaks for website fingerprinting on Android 7. Accuracy based on 20 websites.

procfs file	Property	Accuracy
`/proc/net/netstat`	`IpExt: InNoECTPkts`	82.5%
`/proc/net/netstat`	`IpExt: InOctets`	81.9%
`/proc/net/dev`	`wlan0: Receive packets`	81.9%
`/proc/net/dev`	`wlan0: Received bytes`	78.8%
`/proc/net/dev`	`wlan0: Transmit packets`	77.5%
`/proc/net/netstat`	`IpExt: OutOctets`	73.8%
`/proc/net/dev`	`wlan0: Transmit bytes`	71.9%
`/proc/vmstat`	`nr_shmem`	70.6%
`/proc/vmstat`	`nr_mapped`	64.4%
`/proc/net/sockstat`	`sockets: used`	60.0%

5.2.2 Adversary Model and Evaluation. Similar to the website fingerprinting evaluation we assume that a zero-permission app monitors the identified procfs leaks in the background and exploits these information leaks to infer a user's browsing behavior.

Evaluation. For this proof of concept we established a database of website fingerprints (templates) for the top 20 websites according to alexa.com. We collected 8 samples, *i.e.*, 8 time series for the identified information leaks in Table 6, per website. We combined the identified procfs leaks by means of multi-resource evaluation.

For all gathered samples (time series), ProcHarvester infers visited websites with a high probability. The detailed results for each of the 20 websites are shown in Table 7. Overall, we achieve an average classification rate of 94% based on k-fold cross validation.

Table 7: Classification rates for website fingerprinting by combining the identified information leaks on Android 7. Accuracy based on 8 samples per website.

Website	Precision	Recall
www.360.cn	78%	88%
www.amazon.com	100%	100%
www.baidu.com	100%	100%
www.facebook.com	100%	100%
www.google.com	100%	100%
www.imgur.com	100%	100%
www.instagram.com	88%	88%
www.jd.com	88%	88%
www.linkedin.com	100%	88%
www.live.com	100%	88%
www.netflix.com	100%	88%
www.qq.com	78%	88%
www.reddit.com	100%	100%
www.sina.com.cn	100%	75%
www.sohu.com	78%	88%
www.taobao.com	88%	88%
www.tmall.com	89%	100%
www.vk.com	89%	100%
www.wikipedia.org	100%	100%
www.yahoo.com	100%	100%
Average	94%	93%

Table 8: Excerpt of identified information leaks for website fingerprinting on Android 8. Accuracy based on 20 websites.

procfs file	Property	Accuracy
/proc/net/dev	eth0: Receive packets	80.6%
/proc/net/xt_qtaguid/ iface_stat_all	eth0: rx_bytes (column 6)	80.0%
/proc/net/netstat	IpExt: InOctets	79.4%
/proc/net/sockstat	TCP: mem	78.8%
/proc/net/snmp	Tcp: InSegs	78.8%
/proc/net/dev	eth0: Transmit errs	77.5%
/proc/net/dev	eth0: Receive errs	77.5%
/proc/net/protocols	TCP: memory	75.6%
/proc/net/netstat	IpExt: InNoECTPkts	75.6%
/proc/net/protocols	TCPv6: memory	75.6%
/proc/net/snmp	Ip: InReceives	75.6%
/proc/net/xt_qtaguid/ iface_stat_all	eth0: tx_bytes (column 8)	75.0%
/proc/net/dev	eth0: Transmit packets	75.0%
/proc/net/snmp	Tcp: OutSegs	75.0%
/proc/net/xt_qtaguid/ iface_stat_all	eth0: rx_packets (column 7)	75.0%
/proc/net/snmp	Ip: InDelivers	74.4%
/proc/net/netstat	IpExt: OutOctets	73.8%
/proc/net/snmp	Ip: OutRequests	72.5%
/proc/meminfo	Mapped	55.6%
/proc/net/sockstat	sockets: used	55.0%
/proc/meminfo	Shmem	45.0%
/proc/meminfo	MemFree	42.5%
/proc/meminfo	Active(anon)	36.3%
/proc/meminfo	AnonPages	35.6%
/proc/net/protocols	UNIX: sockets	26.9%
/proc/meminfo	Committed_AS	13.1%

5.3 Analysis and Evaluation on Android 8

Similar to the evaluation on Android 7, ProcHarvester automatically identified information leaks that allow to fingerprint websites on Android 8. The profiling and evaluation are performed exactly as on Android 7. Table 8 provides an excerpt of the information leaks and the corresponding accuracies evaluated for 20 websites on Android 8. By combining the identified information leaks from Table 8 we achieve an average classification rate of 87% based on k-fold cross validation. Table 9 depicts the detailed results for all 20 websites.

6 KEYBOARD GESTURE INFERENCE

We now demonstrate the applicability of ProcHarvester to automatically profile events such as tap, touch, long press, as well as short and long swipe actions on the soft keyboard.

Problem Description. Information about user input gestures (e.g., the length of swipe actions, whether it was a short touch action or a long touch action, etc.) enable powerful follow-up attacks (cf. [24]). Therefore, the Android system prevents applications from directly learning such sensitive information.

6.1 Profiling

In order to profile the procfs for information leaks that reveal sensitive user input activity, we simulate touch actions and touch gestures through ADB commands (`input swipe` and `input tap`).

Table 9: Classification rates for website fingerprinting by combining the identified information leaks on Android 8. Accuracy based on 8 samples per website.

Website	Precision	Recall
www.360.cn	89%	100%
www.amazon.com	80%	100%
www.baidu.com	86%	75%
www.facebook.com	100%	100%
www.google.com	89%	100%
www.imgur.com	70%	88%
www.instagram.com	80%	100%
www.jd.com	71%	62%
www.linkedin.com	80%	100%
www.live.com	89%	100%
www.netflix.com	88%	88%
www.qq.com	100%	25%
www.reddit.com	100%	100%
www.sina.com.cn	100%	75%
www.sohu.com	62%	62%
www.taobao.com	100%	100%
www.tmall.com	100%	88%
www.vk.com	100%	50%
www.wikipedia.org	80%	100%
www.yahoo.com	80%	100%
Average	87%	86%

Note that specific interrupts such as the screen interrupt are only triggered for real touchscreen interactions, which will lead to additional information leaks. However, a complete investigation of information leaks would require the events to be triggered by physically touching the screen, e.g., by a human being, and a fully-fledged attack evaluation is out of scope for this paper. We consider the following events in order to demonstrate the applicability of ProcHarvester:

(1) Short swipe over three soft keys (75 ms)
(2) Long swipe over nine soft keys (300 ms)
(3) Tap character, *i.e.*, keystroke on "a"
(4) Long press character, *i.e.*, long press on "a"
(5) Tap shift key

6.2 Information Leaks on Android 7

6.2.1 Information Leaks. Similar to the experiments in the previous sections, the ProcHarvester Desktop Suite automatically identified several procfs resources that allow to detect the profiled user input events. The evaluation is based on results obtained on the One Plus 3T and the AOSP keyboard.

Figure 6 illustrates plots of the `MDSS` resource for three user input events. Although the traces for "tap character" and "tap shift" look quite similar at first glance, the y-axes have a different scale, which means that these events can be automatically distinguished based on the identified information leak. Table 10 provides an excerpt of the identified information leaks that allow to infer user input actions.

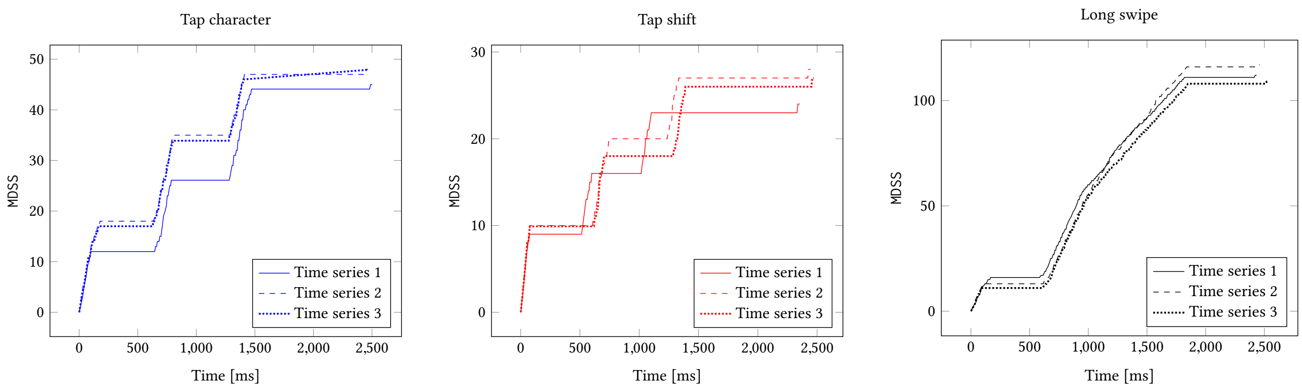

Figure 6: Information leak (MDSS) for "tap character", "tap shift", and "long swipe".

Table 10: Excerpt of identified information leaks for keyboard gestures on Android 7. Accuracy based on 5 gestures.

procfs file	Property	Accuracy
`/proc/interrupts`	`MDSS`	95.0%
`/proc/interrupts`	`kgsl-3do`	86.3%
`/proc/vmstat`	`nr_mapped`	85.0%
`/proc/interrupts`	`Rescheduling interrupts`	77.5%

Table 11: Classification rates for screen gestures by combining the identified information leaks on Android 7. Accuracy based on 10 samples per gesture.

Keyboard gesture	Precision	Recall
Short swipe	100%	100%
Long swipe	100%	100%
Tap character	91%	100%
Long press character	100%	100%
Tap shift	100%	90%
Average	98%	98%

6.2.2 Adversary Model and Evaluation. Similar to the previous adversary models, we assume that the attacker tries to fingerprint user input events. Hence, the malicious application monitors the identified information leaks in order to infer user input events by means of a template attack.

Evaluation. We established a database of screen gesture fingerprints for the above described gestures by collecting 10 samples per gesture. By combining the identified information leaks presented in Table 10, we achieve an average accuracy of 98% based on k-fold cross validation. The detailed results for each of the five gestures can be found in Table 11.

6.3 Analysis and Evaluation on Android 8

Similar to the evaluation on Android 7, ProcHarvester automatically identified information leaks that allow to infer keyboard gestures on Android 8. The profiling and evaluation are performed exactly as on Android 7. Table 12 provides an excerpt of the information leaks and the corresponding accuracies evaluated for different keyboard gestures on Android 8. Although interrupt information (`/proc/interrupts`) and especially the `MDSS` interrupt information is not available anymore on Android 8, ProcHarvester identified many information leaks within `/proc/meminfo` that allow to infer keyboard gestures. Our evaluation shows that the overall accuracy for inferring keyboard gestures decreases, but there are still many information leaks left on Android 8.

Table 12: Excerpt of identified information leaks for keyboard gestures on Android 8. Accuracy based on 5 gestures.

procfs file	Property	Accuracy
`/proc/meminfo`	`Active`	77.5%
`/proc/meminfo`	`Active(anon)`	76.3%
`/proc/meminfo`	`AnonPages`	73.8%
`/proc/meminfo`	`Committed_AS`	72.5%
`/proc/meminfo`	`HighFree`	72.5%
`/proc/meminfo`	`MemFree`	72.5%
`/proc/meminfo`	`MemAvailable`	70.0%
`/proc/meminfo`	`LowFree`	63.8%
`/proc/meminfo`	`VmallocUsed`	62.5%
`/proc/meminfo`	`Mapped`	60.0%
`/proc/meminfo`	`PageTables`	57.5%
`/proc/meminfo`	`KernelStack`	50.0%
`/proc/meminfo`	`Active(file)`	45.0%
`/proc/meminfo`	`Cached`	42.5%
`/proc/meminfo`	`Dirty`	38.8%

By combining the identified information leaks from Table 12 we achieve an average classification rate of 73% based on k-fold cross validation. Table 13 depicts the detailed results for all keyboard gestures. The side channels automatically identified by ProcHarvester are the only known side channels on Android 8.

7 DISCUSSION

We now discuss countermeasures against procfs side-channel attacks and how ProcHarvester can be used to automatically identify procfs leaks before Android updates are shipped to the user. Furthermore, we discuss current limitations as well as the performance of ProcHarvester.

Table 13: Classification rates for screen gestures by combining the identified information leaks on Android 8. Accuracy based on 10 samples per gesture.

Keyboard gesture	Precision	Recall
Short swipe	100%	90%
Long swipe	91%	100%
Tap character	33%	10%
Long press character	91%	100%
Tap shift	50%	80%
Average	73%	76%

7.1 Countermeasures

App Guardian. Zhang et al. [36] proposed a countermeasure to prevent procfs-based side-channel attacks. The main observation is that the attack app needs to run side-by-side with the victim app on the targeted device in order to collect the required side-channel information. Therefore, they developed a third-party application (App Guardian) that should prevent such side-channel attacks. The idea is to detect ongoing side-channel attacks against specific applications by observing, for example, the CPU usage of currently executing applications. Thereby, App Guardian assumes that if the CPU usage of an application increases while an application to be protected is executed, this application might perform a side-channel attack. If such a suspicious application is detected, it will be stopped.

App Guardian [31] has been developed in 2015 and it relies on `getRunningTasks()` as well as `/proc/<pid>/statm` to detect ongoing side-channel attacks. Both resources are not available anymore since Android N (Android 7) and, thus, in its current form App Guardian does not protect against side-channel attacks on recent Android versions. Similarly, Diao et al. [12] observed that App Guardian does not prevent attacks exploiting `/proc/interrupts` on Android 5.1.1. Hence, App Guardian must inevitably be updated for more recent Android versions, which might become a tedious task required for each new Android version.

Restrict Access to procfs Resources. Although Android has already been hardened, our investigations show that more rigorous restrictions for procfs interfaces are essential. The attack surface has already been reduced by continuously restricting access to per-process information (e.g., `/proc/<pid>/`) starting from Android M (Android 6) and also by restricting access to global interrupt information (`/proc/interrupts`) in Android O (Android 8). However, by relying on ProcHarvester we identified several new information leaks that are still publicly available, as they are still considered harmless. ProcHarvester allows to investigate such information leaks more systematically, which is especially interesting for OS designers and OS developers. For instance, although the new Android O (Android 8) restricts access to `/proc/vmstat`, ProcHarvester automatically revealed that the same information is now available in `/proc/meminfo`. Hence, ProcHarvester constitutes a tool for automatically identifying information leaks and is essential for the elimination of procfs information leaks in upcoming Android versions before they are released.

Evaluation of Countermeasures. ProcHarvester can also be used to automatically evaluate newly proposed countermeasures. Especially if countermeasures do not restrict access to a resource but try to protect it, for example, by means of noise injection [32] or by releasing more coarse-grained information [38], ProcHarvester allows developers to automatically evaluate the effectiveness of a proposed countermeasure at a larger scale.

7.2 Limitations

Among many new procfs leaks that allow to infer application launches, visited websites, and keyboard gestures, ProcHarvester also successfully identified already known information leaks automatically. For example, profiling app starts with ProcHarvester revealed the information leaks already exploited by Diao et al. [12] in order to infer application launches. This demonstrates the effectiveness of the proposed ProcHarvester framework. In addition, the generic design of ProcHarvester can be adapted and extended to support the profiling of other events of interest as well. We also demonstrated that information leaks identified by ProcHarvester can be successfully exploited in subsequent side-channel attacks.

A crucial point, however, is that if ProcHarvester does not identify information leaks, it does not necessarily mean that the system is secure and does not leak any information through the procfs. By relying on the generic approach of dynamic time warping, we are able to systematically analyze procfs resources automatically but this does not guarantee that an attacker cannot extract more targeted and specialized features that can be exploited.

Besides, ProcHarvester currently only considers procfs resources that are frequently updated during the profiling of events. This means that it does not consider static information published via the procfs. For example, Chen et al. [7] mentioned that app starts can also be inferred by monitoring `/proc/net/tcp6`, which contains destination IP addresses. This information, however, is static during the profiling and is currently ignored by ProcHarvester.

7.3 Performance

ProcHarvester represents an analysis tool that allows identifying side-channel information leaks automatically. Thus, we neither optimized the Android app in terms of a stealthy attack that aims to reduce the battery consumption, nor did we optimize the analysis in the backend. The DTW-based approach scales quadratically with the number of events since each trace is compared to all other traces in order to determine the inference accuracy.

For example, on an Intel Broadwell 2 GHz with 8 GB of RAM, the analysis takes 2–3 minutes for a set of 20 apps and 14 procfs resources. For a set of 100 apps and 14 procfs resources, this approach takes 49 minutes. Again, we did not optimize the DTW implementation as we did not intend to implement a high-performance attack, but to propose an analysis tool that allows identifying information leaks that can be exploited to launch side-channel attacks.

8 CONCLUSION

In this paper we introduced ProcHarvester, a technique to scan the entire procfs for information leaks in a fully automated fashion. Based on the identified information leaks for application starts, we demonstrated an attack that significantly outperforms state-of-the-art application inference attacks. Furthermore, we demonstrated how ProcHarvester automatically identifies information leaks for other events of interest such as visited websites and keyboard gestures. Our investigations show that the threat of procfs information leaks is omnipresent, and we identified several new side-channel leaks on Android 7 as well as the only procfs information leaks on the recently released Android 8.

Most importantly, ProcHarvester advances the investigation of procfs information leaks. The information gained by using ProcHarvester assists OS designers and OS developers in detecting possible side-channel attacks resulting from information published via the procfs. Based on these insights, we hope that future operating systems will be less susceptible to procfs-based attacks.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

REFERENCES

[1] Android Developers. 2015. Android 6.0 Changes. https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html. (2015). Accessed: August 2017.

[2] Android Developers. n. d.. Android Debug Bridge (ADB). https://developer.android.com/studio/command-line/adb.html. (n. d.). Accessed: August 2017.

[3] Android Developers. n. d.. monkeyrunner. https://developer.android.com/studio/test/monkeyrunner/index.html. (n. d.). Accessed: August 2017.

[4] Android Open Source Project. n. d.. Security-Enhanced Linux in Android. https://source.android.com/security/selinux/. (n. d.). Accessed: August 2017.

[5] Adam J. Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M. Smith. 2012. Practicality of Accelerometer Side Channels on Smartphones. In *Annual Computer Security Applications Conference – ACSAC 2012*. ACM, 41–50.

[6] Liang Cai and Hao Chen. 2011. TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion. In *USENIX Workshop on Hot Topics in Security – HotSec*. USENIX Association.

[7] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *USENIX Security Symposium 2014*. USENIX Association, 1037–1052.

[8] Maximilian Christ, Andreas W. Kempa-Liehr, and Michael Feindt. 2016. Distributed and Parallel Time Series Feature Extraction for Industrial Big Data Applications. *arXiv ePrint Archive, Report 1610.07717* (2016).

[9] Anupam Das, Nikita Borisov, and Matthew Caesar. 2014. Do You Hear What I Hear?: Fingerprinting Smart Devices Through Embedded Acoustic Components. In *Conference on Computer and Communications Security – CCS 2014*. ACM, 441–452.

[10] Anupam Das, Nikita Borisov, and Matthew Caesar. 2016. Tracking Mobile Web Users Through Motion Sensors: Attacks and Defenses. In *Network and Distributed System Security Symposium – NDSS 2016*. The Internet Society.

[11] Sanorita Dey, Nirupam Roy, Wenyuan Xu, Romit Roy Choudhury, and Srihari Nelakuditi. 2014. AccelPrint: Imperfections of Accelerometers Make Smartphones Trackable. In *Network and Distributed System Security Symposium – NDSS 2014*. The Internet Society.

[12] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. 2016. No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. In *IEEE Symposium on Security and Privacy – S&P 2016*. IEEE Computer Society, 414–432.

[13] Jun Han, Emmanuel Owusu, Le T. Nguyen, Adrian Perrig, and Joy Zhang. 2012. ACComplice: Location Inference Using Accelerometers on Smartphones. In *International Conference on Communication Systems and Networks – COMSNETS 2012*. IEEE, 1–9.

[14] Suman Jana and Vitaly Shmatikov. 2012. Memento: Learning Secrets from Process Footprints. In *IEEE Symposium on Security and Privacy – S&P 2012*. IEEE Computer Society, 143–157.

[15] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium 2016*. USENIX Association, 549–564.

[16] Maryam Mehrnezhad, Ehsan Toreini, Siamak Fayyaz Shahandashti, and Feng Hao. 2016. TouchSignatures: Identification of User Touch Actions and PINs Based on Mobile Sensor Data via JavaScript. *J. Inf. Sec. Appl.* 26 (2016), 23–38.

[17] Yan Michalevsky, Dan Boneh, and Gabi Nakibly. 2014. Gyrophone: Recognizing Speech from Gyroscope Signals. In *USENIX Security Symposium 2014*. USENIX Association, 1053–1067.

[18] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. 2015. PowerSpy: Location Tracking Using Mobile Device Power Analysis. In *USENIX Security Symposium 2015*. USENIX Association, 785–800.

[19] Emiliano Miluzzo, Alexander Varshavsky, Suhrid Balakrishnan, and Romit Roy Choudhury. 2012. Tapprints: Your Finger Taps Have Fingerprints. In *Mobile Systems – MobiSys 2012*. ACM, 323–336.

[20] Meinard Müller. 2007. *Dynamic Time Warping*. Springer, 69–84. https://doi.org/10.1007/978-3-540-74048-3

[21] Sashank Narain, Triet D. Vo-Huu, Kenneth Block, and Guevara Noubir. 2016. Inferring User Routes and Locations Using Zero-Permission Mobile Sensors. In *IEEE Symposium on Security and Privacy – S&P 2016*. IEEE Computer Society, 397–413.

[22] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. 2012. ACCessory: Password Inference Using Accelerometers on Smartphones. In *Mobile Computing Systems and Applications – HotMobile 2012*. ACM, 9.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[24] Laurent Simon, Wenduan Xu, and Ross Anderson. 2016. Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards. *PoPETs* 2016 (2016), 136–154.

[25] Raphael Spreitzer. 2014. PIN Skimming: Exploiting the Ambient-Light Sensor in Mobile Devices. In *Security and Privacy in Smartphones & Mobile Devices – SPSM@CCS*. ACM, 51–62.

[26] Raphael Spreitzer and Benoît Gérard. 2014. Towards More Practical Time-Driven Cache Attacks. In *Information Security Theory and Practice – WISTP 2014 (LNCS)*, Vol. 8501. Springer, 24–39.

[27] Raphael Spreitzer, Simone Griesmayr, Thomas Korak, and Stefan Mangard. 2016. Exploiting Data-Usage Statistics for Website Fingerprinting Attacks on Android. In *Security and Privacy in Wireless and Mobile Networks – WISEC 2016*. ACM, 49–60.

[28] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. 2018. Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices. *IEEE Communications Surveys and Tutorials* 20 (2018), 465–488.

[29] Raphael Spreitzer and Thomas Plos. 2013. Cache-Access Pattern Attack on Disaligned AES T-Tables. In *Constructive Side-Channel Analysis and Secure Design – COSADE 2013 (LNCS)*, Vol. 7864. Springer, 200–214.

[30] Raphael Spreitzer and Thomas Plos. 2013. On the Applicability of Time-Driven Cache Attacks on Mobile Devices. In *Network and System Security – NSS 2013 (LNCS)*, Vol. 7873. Springer, 656–662.

[31] System Security Lab, Indiana University. 2015. App Guardian. https://play.google.com/store/apps/details?id=edu.iub.seclab.appguardian. (2015). Accessed: August 2017.

[32] Qiuyu Xiao, Michael K. Reiter, and Yinqian Zhang. 2015. Mitigating Storage Side Channels Using Statistical Privacy Mechanisms. In *Conference on Computer and Communications Security – CCS 2015*. ACM, 1582–1594.

[33] Zhi Xu, Kun Bai, and Sencun Zhu. 2012. TapLogger: Inferring User Inputs on Smartphone Touchscreens Using On-Board Motion Sensors. In *Security and Privacy in Wireless and Mobile Networks – WISEC 2012*. ACM, 113–124.

[34] Lin Yan, Yao Guo, Xiangqun Chen, and Hong Mei. 2015. A Study on Power Side Channels on Mobile Devices. In *Symposium of Internetware – Internetware 2015*. ACM, 30–38.

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

[36] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiao-yong Zhou, and XiaoFeng Wang. 2015. Leave Me Alone: App-Level Protection against Runtime Information Gathering on Android. In *IEEE Symposium on Security and Privacy – S&P 2015*. IEEE Computer Society, 915–930.

[37] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. 2016. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *Conference on Computer and Communications Security – CCS 2016*. ACM, 858–870.

[38] Xiao-yong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A. Gunter, and Klara Nahrstedt. 2013. Identity, Location, Disease and More: Inferring Your Secrets From Android Public Resources. In *Conference on Computer and Communications Security – CCS 2013*. ACM, 1017–1028.

[39] Zhe Zhou, Wenrui Diao, Xiangyu Liu, and Kehuan Zhang. 2014. Acoustic Fingerprinting Revisited: Generate Stable Device ID Stealthily with Inaudible Sound. In *Conference on Computer and Communications Security – CCS 2014*. ACM, 429–440.

A CONSIDERED ANDROID APPLICATIONS ON ANDROID 7

Table 14 shows the 100 apps used in the evaluation of app cold start detection in Section 4. Precision and recall are determined based on 10 samples for each application. For comparison reasons, we aimed to rely on the set of 100 apps used by Diao et al. [12]. However, only 65 of these apps have been available at the time of writing this paper and, thus, we replaced the remaining 35 apps with common apps from the Google Play store.

Table 14: Applications used for app inference evaluation. Evaluation is based on 10 samples per app.

Package name	Precision	Recall
air.com.hoimi.MathxMath	90%	90%
air.com.hypah.io.slither	100%	100%
at.DiTronic.androidgroup.randomgallery	100%	90%
bbc.mobile.news.ww	100%	80%
cmb.pb	90%	90%
cn.etouch.ecalendar.longshi2	90%	90%
com.Kingdee.Express	91%	100%
com.Slack	100%	80%
com.aastocks.dzh	100%	100%
com.airbnb.android	100%	90%
com.ajnsnewmedia.kitchenstories	100%	90%
com.amazon.mShop.android.shopping	77%	100%
com.android.chrome	91%	100%
com.android.vending	100%	100%
com.antutu.ABenchMark	100%	100%
com.baidu.baidutranslate	100%	80%
com.baidu.searchbox	100%	100%
com.bankofamerica.cashpromobile	100%	100%
com.booking	91%	100%
com.chase.sig.android	100%	90%
com.citrix.saas.gotowebinar	77%	100%
com.cnn.mobile.android.phone	100%	100%
com.coolmobilesolution.fastscannerfree	100%	100%
com.csst.ecdict	83%	100%
com.dewmobile.kuaiya.play	91%	100%
com.douban.frodo	100%	100%
com.dropbox.android	100%	90%
com.ebay.mobile	100%	100%
com.facebook.katana	77%	100%
com.facebook.orca	100%	80%
com.facebook.pages.app	100%	80%
com.facebook.work	100%	100%
com.google.android.apps.docs	83%	100%
com.google.android.apps.photos	100%	90%
com.google.android.deskclock	100%	100%
com.google.android.gm	100%	100%
com.google.android.keep	100%	100%
com.google.android.music	100%	100%
com.google.android.street	100%	100%
com.google.android.youtube	100%	100%
com.groupon.redemption	90%	90%
com.healthagen.iTriage	100%	90%
com.hket.android.ctjobs	86%	60%
com.hse28.hse28_2	100%	90%
com.htsu.hsbcpersonalbanking	100%	100%
com.imdb.mobile	100%	100%
com.indeed.android.jobsearch	100%	100%
com.instagram.android	100%	80%
com.intsig.BCRLite	100%	100%
com.intsig.camscanner	100%	100%
com.isis_papyrus.raiffeisen_pay_eyewdg	91%	100%
com.jobmarket.android	91%	100%
com.jobsdb	100%	90%
com.king.candycrushsaga	100%	100%
com.kpmoney.android	91%	100%
com.lenovo.anyshare.gps	100%	100%
com.linkedin.android.jobs.jobseeker	91%	100%
com.magisto	100%	100%
com.malangstudio.alarmmon	100%	100%
com.medscape.android	100%	100%
com.microsoft.hyperlapsemobile	100%	100%
com.microsoft.rdc.android	91%	100%
com.miniclip.agar.io	100%	100%
com.mmg.theoverlander	90%	90%
com.mobisystems.office	91%	100%
com.money.on	100%	100%
com.mt.mtxx.mtxx	100%	100%
com.mtel.androidbea	100%	100%
com.mysugr.android.companion	100%	100%
com.netflix.mediaclient	100%	100%
com.nianticlabs.pokemongo	100%	100%
com.nuthon.centaline	100%	100%
com.openrice.android	100%	90%
com.paypal.android.p2pmobile	91%	100%
com.priceline.android.negotiator	91%	100%
com.roidapp.photogrid	100%	100%
com.sankuai.movie	100%	100%
com.scb.breezebanking.hk	100%	100%
com.skype.raider	100%	100%
com.smartwho.SmartAllCurrencyConverter	91%	100%
com.smule.singandroid	100%	60%
com.snapchat.android	91%	100%
com.sometimeswefly.littlealchemy	100%	100%
com.spotify.music	100%	100%
com.surpax.ledflashlight.panel	100%	100%
com.ted.android	91%	100%
com.tinder	100%	100%
com.tripadvisor.tripadvisor	100%	90%
com.twitter.android	100%	80%
com.whatsapp	71%	100%
com.zhihu.android	100%	100%
ctrip.android.view	100%	100%
io.appsoluteright.hkexChecker	100%	100%
io.silvrr.silvrrwallet.hk	100%	100%
jp.united.app.kanahei.money	83%	100%
org.telegram.messenger	89%	80%
sina.mobile.tianqitong	100%	100%
tools.bmirechner	100%	100%
tv.danmaku.bili	100%	100%
tw.com.off.hkradio	100%	90%
Average	96%	96%

Continued on next column

Continued from previous column

Table 15 shows the 20 apps used in the evaluation of app resumes in Section 4. Precision and recall are determined based on 10 samples for each app. These 20 apps have been randomly selected from the set of 100 apps in Table 14. Note that this set of 20 apps has been generated once and re-used for the evaluations on Android 7.

Table 15: Applications used for app resume inference evaluation. Evaluation is based on 10 samples per app.

Package name	Precision	Recall
at.DiTronic.androidgroup.randomgallery	100%	90%
com.android.chrome	100%	90%
com.android.vending	55%	60%
com.dropbox.android	64%	70%
com.facebook.orca	100%	90%
com.google.android.apps.photos	73%	80%
com.google.android.gm	70%	70%
com.google.android.music	33%	20%
com.instagram.android	100%	100%
com.isis_papyrus.raiffeisen_pay_eyewdg	75%	90%
com.lenovo.anyshare.gps	100%	100%
com.paypal.android.p2pmobile	83%	100%
com.scb.breezebanking.hk	100%	100%
com.snapchat.android	83%	100%
com.sometimeswefly.littlealchemy	100%	100%
com.ted.android	80%	80%
com.whatsapp	100%	80%
io.silvrr.silvrrwallet.hk	100%	100%
org.telegram.messenger	100%	100%
tv.danmaku.bili	100%	100%
Average	86%	86%

Table 16 shows the 20 apps used in the mixed app inference evaluation of cold starts and resumes in Section 4. Precision and recall are determined based on 16 samples for each application, *i.e.*, 8 samples for cold starts and 8 samples for resumes.

Table 16: Applications used for mixed app inference evaluation (cold starts and resumes). Evaluation is based on 16 samples per app.

Package name	Precision	Recall
at.DiTronic.androidgroup.randomgallery	100%	75%
com.android.chrome	94%	100%
com.android.vending	69%	56%
com.dropbox.android	80%	75%
com.facebook.orca	93%	81%
com.google.android.apps.photos	86%	75%
com.google.android.gm	92%	75%
com.google.android.music	85%	69%
com.isis_papyrus.raiffeisen_pay_eyewdg	79%	94%
com.lenovo.anyshare.gps	76%	100%
com.paypal.android.p2pmobile	94%	100%
com.scb.breezebanking.hk	100%	100%
com.snapchat.android	100%	94%
com.sometimeswefly.littlealchemy	100%	100%
com.ted.android	100%	100%
com.whatsapp	83%	94%
io.silvrr.silvrrwallet.hk	80%	100%
org.cyanogenmod.snap	100%	100%
org.telegram.messenger	84%	100%
tv.danmaku.bili	100%	100%
Average	90%	89%

Table 17 shows the 20 apps used in the manual evaluation of cold start detection in Section 4. Precision and recall are determined based on 10 samples for each application.

Table 17: Applications used for app inference evaluation triggered manually (w/o ADB). Evaluation is based on 10 samples per app.

Package name	Precision	Recall
at.DiTronic.androidgroup.randomgallery	100%	100%
com.android.chrome	100%	80%
com.android.vending	100%	100%
com.dropbox.android	100%	100%
com.facebook.orca	83%	100%
com.google.android.apps.photos	100%	100%
com.google.android.gm	100%	100%
com.google.android.music	100%	100%
com.instagram.android	100%	90%
com.isis_papyrus.raiffeisen_pay_eyewdg	91%	100%
com.lenovo.anyshare.gps	100%	100%
com.paypal.android.p2pmobile	100%	100%
com.scb.breezebanking.hk	100%	100%
com.snapchat.android	100%	100%
com.sometimeswefly.littlealchemy	100%	100%
com.ted.android	100%	100%
com.whatsapp	100%	90%
io.silvrr.silvrrwallet.hk	100%	100%
org.telegram.messenger	91%	100%
tv.danmaku.bili	100%	100%
Average	98%	98%

B CONSIDERED ANDROID APPLICATIONS ON ANDROID 8

Table 18 shows the 20 apps used for the evaluation of cold start detection on Android 8. Precision and recall are determined based on 10 samples for each application.

Table 18: Applications used for app inference evaluation on Android 8. Evaluation is based on 10 samples per app.

Package name	Precision	Recall
air.com.hypah.io.slither	100%	100%
at.DiTronic.androidgroup.randomgallery	100%	100%
com.Slack	90%	90%
com.android.chrome	100%	90%
com.android.vending	75%	90%
com.bankofamerica.cashpromobile	100%	100%
com.dropbox.android	100%	80%
com.ebay.mobile	86%	60%
com.google.android.apps.docs	100%	100%
com.google.android.apps.photos	100%	80%
com.google.android.gm	82%	90%
com.google.android.keep	75%	30%
com.google.android.music	90%	90%
com.scb.breezebanking.hk	100%	100%
com.sololearn.cplusplus	71%	100%
com.sometimeswefly.littlealchemy	91%	100%
com.ted.android.conference	56%	90%
com.twitter.android	100%	100%
com.whatsapp	75%	60%
org.catrobat.catroid	50%	60%
Average	87%	85%