No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis

Wenrui Diao*, Xiangyu Liu*, Zhou Li†, and Kehuan Zhang*
*Department of Information Engineering, The Chinese University of Hong Kong
Email: {dw013, lx012, khzhang}@ie.cuhk.edu.hk
†IEEE Member
Email: lzcarl@gmail.com

***Abstract*—Many new specialized hardware components have been integrated into Android smartphones to improve mobility and usability, such as touchscreen, Bluetooth module, and NFC controller. At the system level, the kernel of Android is built on Linux and inherits its device management mechanisms. However, the security implications surfaced from the integration of new hardware components and the tailored Linux kernel are not fully understood. In this paper, we make the first attempt to evaluate such implications. As a result, we identify a critical information leakage channel from the interrupt handling mechanism, which can be exploited to launch inference attacks without any permission. On Android, all reported interrupts are counted by Linux kernel and the statistical information is logged in a system file `/proc/interrupts`, which is public to any process. Such statistical information reveals the running status of all integrated devices, and could be exploited by attackers to infer sensitive information passing through them. To assess this new threat, we propose a general attack approach – interrupt timing analysis and apply it to interrupt logs. As showcases, we present two concrete inference attacks against user's unlock pattern and foreground app status respectively. Through analyzing the interrupt time series produced from touchscreen controller, attacker's chance of cracking user's unlock pattern is increased substantially. The interrupt time series produced from Display Sub-System reveals unique UI refreshing patterns and could be leveraged as fingerprints to identify the app running in the foreground. Such information can serve as the stepping stone for the subsequent phishing attacks. The experiment results suggest our inference attacks are highly effective, and the risks should be mitigated immediately.**

***Index Terms*—hardware interrupt; timing analysis; procfs;**

I. Introduction

Smartphone plays the important role of personal assistant and data container in people's daily life. Different from the traditional desktop platforms, mobile OSes need to suffice the new requirements of mobility and usability with limited computing resources. New specialized hardware components, e.g., touchscreen and NFC transmitter, are manufactured and integrated to this end.

Previous research investigated the security threats coming from particular hardware components, like accelerometer [1], [2] and camera [3], [4]. These attacks are mainly based on reading data directly generated by the targeted hardware. However, none of them looked into the threats introduced by the highly tailored software components, especially from the angle of kernel. In this paper, we make the first attempt to evaluate the security implications of the integration of specialized hardware and tailored kernel on Android.

Hardware Interrupt. As the most popular mobile platform, Android is built on Linux kernel and enhanced to adapt to the requirement of mobility. Android also inherits the *interrupt mechanism* from Linux, which is designed for the efficient communication between the CPU and external devices. When new hardware events (e.g., user touching the screen) come, the corresponding hardware device (e.g., touchscreen controller) sends a signal to ask OS for immediate processing. As a response, the CPU alters the sequence of instructions in execution to handle this event with high priority.

Our Findings. All reported interrupts are counted by OS, and the statistical information is stored in a system file `/proc/interrupts` (Fig. 2 shows an example), which is public to any process. Such information reflects the real-time running status of devices and it could be exploited by attackers to infer information that passes through, including user's sensitive data and interactions with the devices. In this paper, we propose a novel attack approach – *interrupt timing analysis*. Through analyzing the time series of interrupts occurred for a particular device, user's associated sensitive information could be inferred by the attacker without any permission.

The root cause of this new security threat comes from the ill-conceived integration of specialized hardware components and tailored kernel. Unlike the traditional desktop platforms, smartphone is equipped with many peripheral devices, such as touchscreen, Bluetooth module, NFC controller. These newly included devices usually interact with the user directly and bring new kinds of interrupts, which means new attack channels. However, Android still uses the built-in method of Linux kernel for interrupt auditing without any change. The interrupt information channel was once mentioned in previous work [5], [6], but they doubted that it could be used for effective attacks and no concrete attack method or result was given. On Android, we present two concrete inference attacks

through interrupt timing analysis, against user's unlock pattern and foreground app respectively.

Inferring Unlock Pattern. Touchscreen is an indispensable component for nearly all smartphones. A lot of user's sensitive information passes to the system through the movement on touchscreen. We use the Android graphical password scheme – unlock pattern as an example to demonstrate the feasibility of our proposed interrupt timing analysis. We find the finger movements on the screen correlates to the amount of interrupt occurred for touchscreen controller. Specific to unlock pattern, the interrupt amount reflects the length of swipe line indirectly. After gathering its time series, we leverage Hidden Markov Model and probabilistic analysis to infer the possible unlock patterns. Our experimental study shows, even without any knowledge about the victim (i.e., the user could select any pattern from all 389,112 combinations), our attack could reduce more than 90 % search space for just one guess.

Inferring Foreground App. The information about app running in the foreground is quite sensitive and should be kept away from unauthorized apps to prevent phishing attacks. Starting from Android 5.0, Google has enforced a system-level permission `REAL_GET_TASKS` [7] to prevent such information leaking to third-party apps, which nonetheless can be bypassed by our attack. We observe that, while launching an app, the foreground UI is continuously refreshed. The refreshing patterns of apps during launching could be used to identify this app. Also, in the course of UI refreshing, the interrupts for Display Sub-System (DSS) occur with the same frequency. It motivates us to model the UI refreshing process through interrupt time series and detect the foreground app. Combined with machine learning techniques, we achieve such attack without any permission, and the result shows an attacker has 87 % success rate to identify the foreground app from a set of 100 candidates within one attempt.

Contributions. We summarize this paper's contributions as below:

- *New Attack Surface and Approach.* We discover that the interrupt log file (`/proc/interrupts`) of Android could leak user's sensitive information. To exploit such information, we propose a new general attack approach – interrupt timing analysis.
- *New Inference Attacks.* We present two practical inference attacks to infer user's unlock pattern and the app running in the foreground. The attack channels are based on the interrupt time series for touchscreen controller and Display Sub-System.
- *New Techniques.* A set of novel schemes are developed to model the inference target, including unlock pattern modeling, gram transition inference, improved time series similarity calculation, etc.
- *Implementation and Evaluation.* We implemented the attack prototype apps and evaluated them under the real-world settings. Our experimental studies demonstrate that attacking through interrupt timing side-channel is feasible and highly effective.

Roadmap. The rest of the paper is organized as follows: Section II introduces the background of hardware interrupt and explains why information could be leaked from interrupts on Android. Section III outlines the high-level ideas of our two concrete attacks. Section IV shows our attack on inferring unlock pattern. Inferring the foreground app is introduced in Section V. In Section VI, we discuss the leaks from other interrupts and propose some defense solutions. Related works are reviewed in Section VII. Section VIII concludes this paper.

II. Background

Hardware interrupt was introduced as an efficient mechanism for the communication between high-speed CPU and low-speed external devices since the early age of modern computer. This mechanism is embraced by all mainstream computing platforms, including the mobile ones like Android. In this section, we briefly overview the design of hardware interrupt and Android, and then explain why the leaks from interrupts on Android could lead to inference attacks.

A. Hardware Interrupt

In modern computing architecture, CPU is responsible for managing the connected hardware devices and initiating the handling procedures defined by operating system (OS) for different hardware events. Hardware interrupt mechanism is proposed to enable timely event management when one CPU has to serve many devices (e.g., mouse, keyboard, and network card). When a device requires immediate attention from OS, e.g., when the volume key is pressed or keyboard is typed, an electronic signal will be emitted from the device [8]. Such electronic signal is called *Interrupt Request (IRQ)* and is passed to the corresponding programmable interrupt controller (PIC) through IRQ lines. PIC is a hardware circuit which bridges I/O devices and CPU. When it receives an IRQ, it will notify CPU to process this IRQ immediately. In response, CPU will halt the current execution thread, preserve the execution context, and invoke the registered *interrupt handler*. Such a process that alters the sequence of instructions executed by CPU is called *interrupt*. When the execution of the interrupt handler is completed, the preserved context is restored and halted execution is resumed. This interrupt mechanism is particularly useful for handling hardware I/O events which are usually urgent but triggered nondeterministically [9].

B. Android Platform

In this work, we investigate the security issues regarding hardware interrupt on Android platform. Android system is built on Linux kernel, and new layers are introduced in addition to suffice the functionality requirements for mobile devices [10], [11]. The whole architecture can be sliced into 6 layers, which can be further classified into two categories based on their degrees of dependence on hardware. Fig. 1 illustrates the layers of Android system, and the each layer is briefly described below:

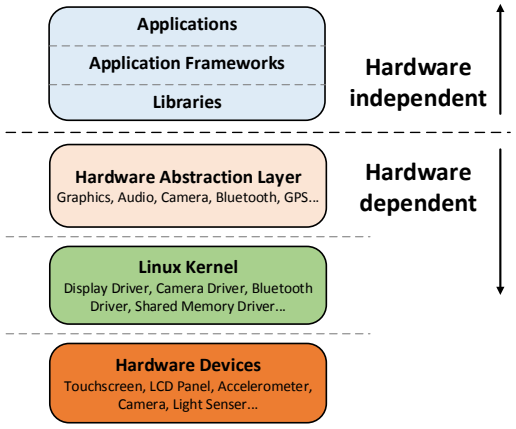

Fig. 1. Android layered framework

1) Hardware Dependent Layers:

- **Hardware Devices** are the physical components of an Android smartphone, including touchscreen, LCD panel, camera, etc.
- **Linux Kernel** is the foundation of Android system. Compared with the original Linux kernel, a set of "kernel enhancements" are patched to provide tailored support for Android system. For instance, an Android-specific mechanism *Binder* is integrated into the kernel to facilitate interprocess communication [12].
- **Hardware Abstraction Layer (HAL)** defines the functional interfaces that are required to be implemented by hardware device drivers. Through this layer, applications and system libraries can operate hardware devices manufactured by different vendors using unified APIs.

2) Hardware Independent Layers:

- **Libraries** layer provides libraries written in native code (C/C++) that directly access Linux kernel and HAL. In addition, Android Runtime, including the mobile application (app) container Dalvik VM, is implemented within this layer.
- **Application Frameworks** include a set of system services that can be shared and reused by mobile apps. For example, *Content Provider* allows data publishing and sharing between apps.
- **Applications** layer stacks the mobile apps. It comprises both system apps which are shipped together with Android OS and third-party apps installed by user.

Under the Android framework, hardware interrupt is raised from the hardware devices layer and responded by the Linux kernel layer. The interrupt flows initiated from specific devices are shown in Section IV-A and V-A.

C. Information Leaks from Interrupts on Android

The framework for interrupt processing and auditing on Android resembles other Linux-based systems. When an IRQ is issued and handled by CPU, it is logged by a system counter and the statistical information is stored in a system file `/proc/interrupts`. The access attributes for this file is `-r--r--r--`, which means it can be read by any process, no matter if it belongs to system apps or third-party apps. Fig. 2 shows the partial content of the file dumped from Google Nexus 6 with Android 5.1.1 installed. The first column lists the unique IRQ (line) numbers. Each number is reserved by OS for one type of event and is associated with one interrupt handler. The following columns show how many interrupts have been issued to each CPU core since the starting of system (power on for Android). Since Google Nexus 6 is equipped with a 4-core CPU (Qualcomm Snapdragon 805), the interrupt counts are listed in 4 columns. The number of columns varies depending on the number of active CPU cores. In general, Linux kernel prefers to handle interrupts on the first CPU core in order to maximize cache locality [8]. As shown in Fig. 2, the column `CPU0` has the largest interrupt count for most IRQ numbers. The penultimate column shows the name of PIC assigned for handling one type of IRQ. One PIC could be shared by multiple devices for transmitting IRQ. The devices that send out the specific IRQ are listed in the last column.

Since the statistical information of interrupt reflects the running status of devices and is also public, it could be abused by attackers to infer the user's actions on targeted devices, causing information leaks. Previous research has mentioned the potential privacy risks related to interrupt [5], [6], but none of them presented concrete attacks leveraging this channel. In particular, Jana et al. [5] stated that "*the feasibility of this attack remains open*". For the traditional desktop environment, though more than 10 different types of interrupts can be monitored, most of the associated devices do not directly interact with users, e.g., system timer. For the remaining ones, the information leaked is rather limited. As an example, the interrupt from audio codec could leak whether an earphone is plugged, which, however, cannot be used to infer more sensitive information - the played sounds. What's more, the interrupt counter is used globally rather than dedicated to a single process, which makes the sensitive information hard to be discerned.

The development of modern mobile phones, however, offers new alternatives to exploit interrupt information for attacks. The set of integrated peripheral devices on smartphones keeps growing to meet user's new expectations, and each new device brings in new kind of interrupt (e.g., NFC controller). For the Google Nexus 6 phone we tested, already more than 100 IRQ numbers are reserved by OS. Most of the newly included devices directly interact with the user, and the potential attack surface is significantly broadened. Another favorable condition for attacks is that applications run less parallel in mobile system than in desktop system. Especially for Android phones, only one application is allowed to run in the foreground. Therefore, the signals from user's actions or applications of interest are more distinguishable. As shown in our concrete attacks, inference attack abusing leaks from interrupt is not only feasible on Android but can also lead to grave security and privacy issues, i.e., leaks of unlock patterns and application running status.

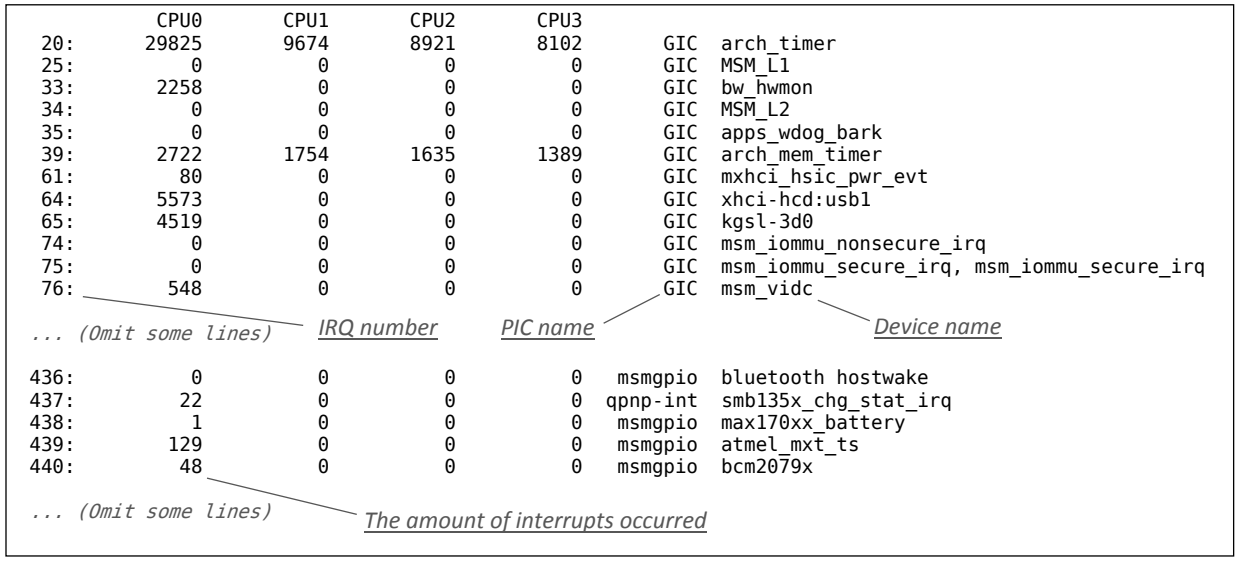

Fig. 2. Example of `/proc/interrupts`, dumped from Google Nexus 6.

Interrupt on Other Platforms. To fully understand the attack surface, we also investigate the attack feasibility on other platforms. In Mac OS X / iOS environment, a similar interrupt mechanism is implemented, but the statistical information of interrupts is not exposed to the processes (no public `/proc` filesystem is available due to the different kernel-level implementation derived from BSD) [13]. On Microsoft Windows platforms (both desktop and mobile), the design principle of interrupt handling is similar, but the implementation is quite different. For instance, the Trap Dispatching mechanism [14] is incorporated to achieve more flexible interrupt processing. On the other hand, different from Mac OS X / iOS platforms, the statistics of interrupt are public. Command-line tools such as `Xperf` [15] can be used to retrieve such information, which is usable for the interrupt attacks.

Software Interrupt. Another kind of interrupt is software interrupt, which is used by programs for immediate communication with CPU. Software interrupt is triggered under two scenarios [9]: 1) an exception (or trap) which cannot be handled by the program alone is raised. 2) special instruction causing interrupt is executed (e.g., to request data from disk controller). The corresponding statistics are logged in a public file `/proc/softirqs`. Yet, whether software interrupt can be used for inference attacks is unclear, since the information is more coarse-grained (aggregated from all processes) and less user-centric. Therefore, we focus on hardware interrupt in this work.

III. Attack Overview

Through examining the interrupt statistics on Android, we identified two types of interrupts that are both tied to user's behaviors and showing distinguishable patterns according to different user's actions. The first type is produced from touchscreen when pressed and released by the user, while the second one is generated accompanying with UI refreshing. By continuously monitoring these interrupt statistics, we show it is feasible to infer user's unlock pattern and the app started by user. In this section, we introduce the mechanisms regarding unlock pattern and UI refreshing together with the high-level ideas of our attacks.

A. Inferring Unlock Pattern

Touchscreen becomes an indispensable component for nearly all smartphones today. A large amount of user's sensitive information passes through the touchscreen, including text messages and unlock pattern. The secrets can be indirectly inferred from the interrupt emitted from touchscreen, and we use unlock patterns as an attack showcase.

Android Unlock Pattern. Unlock pattern is incorporated into Android as an alternative to overcome the usability issue involving traditional authentication schemes, like text-based password. When a user intends to unlock her phone, a 3×3 matrix (totally 9 *dots*) is displayed on the screen, and the user is required to draw her unlock pattern through a series of lines which connect the dots in a certain order (we call them *swipe lines*). Compared to the traditional authentication schemes, unlock pattern is easier to remember and input. Therefore, it is widely adopted by customers [16]. Fig. 3 shows the lock UI on Android 5.1.1 (AOSP) and one unlock pattern.

Each dot in the lock UI is mapped to a number (see Fig. 4), and the number sequence for an unlock pattern is called *pattern password*. For instance, the pattern password of the unlock pattern illustrated in Fig. 3 is `41235789`. For the same *geometric shape* of unlock pattern, drawing in different directions (e.g., `14789` v.s. `98741`) results in different pattern passwords.

A valid pattern lock should satisfy 4 requirements below:

1) At least 4 dots must be used.
2) At most 9 dots can be used.

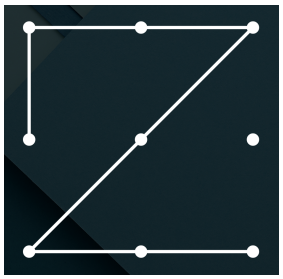

Fig. 3. UI for inputting unlock pattern on AOSP Android 5.1.1.

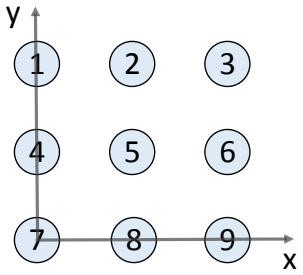

Fig. 4. Unlock pattern modeling.

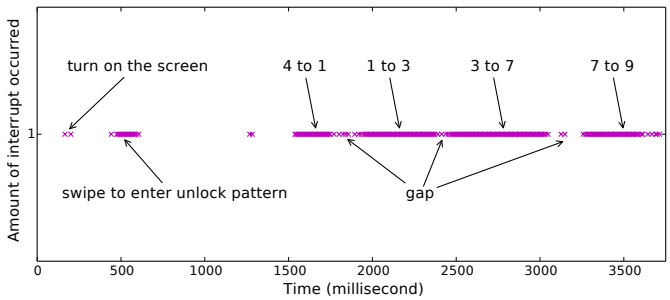

Fig. 5. Interrupt time series for pattern `41235789` inputted by a participant. Within 1 ms, at most 1 interrupt can be observed.

3) No dot can be used more than once.
4) Only straight line is allowed and the dots not visited before cannot be jumped over.

Even after the above restrictions are applied, there still exist 389,112 valid combinations [17], making the chance for successful brute-force attacks very low. Android also enforces strong protection mechanisms to keep the pattern password out of the reach of adversary. The pattern password is stored at `/data/system/gesture.key` and is only accessible to the OS itself (file attribute: `-rw-------`). The raw data of user's touch traces are recorded in `/dev/input/deviceX` (`X` is an integer and varies for different phones). This file is only open to system-level processes belonging to the `input` user group. Therefore, it is impossible to directly steal the password unless the root privilege is obtained. Instead, our attack aims to infer the user's unlock pattern through a zero-permission third-party app which is much stealthier.

Unlock Pattern and Interrupt. When user's finger touches and swipes on the screen, a sequence of interrupts will be produced, which is similar to moving the mouse on desktop. Particularly, in this case, different lines could result in different interrupt sequences and a gap could be observed between lines' interrupts. As an example illustrated in Fig. 5, when drawing the pattern `41235789`, the amount of interrupts observed from the line 3 to 7 is more than other lines, and gaps occur at dot 1, 3 and 7.

The correlation between interrupts and finger movement motivates us to model unlock pattern through interrupt timing analysis. When the interrupt time series is gathered, we first seek to segment it into sequences of incessant interrupts. For each sequence, we look into the observed amount of interrupts and map it to one type of swipe line (e.g., short v.s. long). By combining the inferred swipe lines, we are able to recover the pattern password with decent probabilities.

B. Inferring foreground app

The information about the app running in the foreground (or *foreground app*) should be kept away from unauthorized apps to prevent phishing attacks. However, the name of the foreground app can be speculated through interrupt timing analysis. Furthermore, we found certain app's activity exhibits distinctive interrupt pattern, causing privacy leaks.

Foreground App Detection. The information of foreground app is considered sensitive. If leaked, malicious apps can exploit it for phishing attacks. For example, when a malicious app identifies that an e-banking app is started, it can immediately pop up a phishing window covering the foreground with the same UI as the login page of the e-banking app and fool the user to type her credentials in the fake UI [18], [19]. To mitigate this threat, Google mediated the access to such information through permissions. For the early versions of Android, an app with `GET_TASKS` permission granted can invoke the `getRunningTasks()` API to learn the foreground app. Since Android 5.0, Google replaced the `GET_TASKS` permission with a nonpublic system-level permission `REAL_GET_TASKS` [7], which blocks the access from third-party apps.

UI Refreshing and Interrupts. When an app is launched by the user, a system service `ActivityManagerService` will start the default main activity of this app and then the functions `onCreate()`, `OnStart()`, and `OnResume()` are executed sequentially for app loading. The process also happens for Activity transition after running. At the low level, the foreground UI is continuously refreshed during these processes. An app could choose the way UI is refreshed, and we elaborate three popular UI refreshing related techniques below:

- *Splash Screen.* It is usually shown when an app is started by the user. In most cases, a splash screen displays the promotion information (e.g., logo) or running status (e.g., network connectivity or data loading progress).
- *Asynchronous Loading.* When large data is being loaded from remote servers or internal storage during UI transition, asynchronous loading technique is leveraged, which separates UI rendering and data loading into isolated threads. Therefore, UI is continuously updated instead of being blocked during data loading process.
- *Animation.* App developers can choose animation effects during activity switching. Popular effects include fading in, zooming, wiping, etc. Besides, the animation is also used for rendering control objects such as `ImageButton`.

Fig. 6 shows the launching process of a popular file sharing app SHAREit [20]. The app first displays a splash screen and

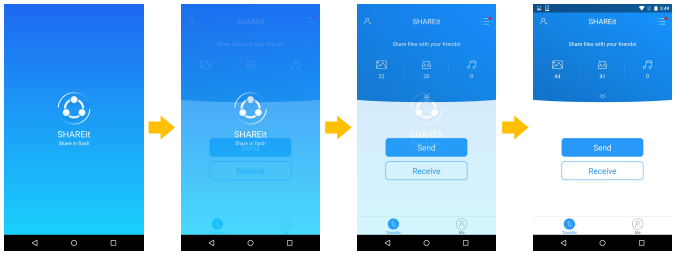

Fig. 6. The launching process of SHAREit. The interrupt time series is shown in the left top of Fig. 7.

the gradually transit the UI to the main Activity with fading animation effect.

In the course of UI refreshing, *Display Sub-System (DSS)* keeps notifying Android system through sending IRQs, and our observation suggests the UI layout and refreshing strategies usually yield distinct interrupt time series. One example is splash screen, for which showing static image will generate much fewer interrupts than showing animation. An app doing asynchronous loading usually refreshes UI more frequently, which leads to continuous occurrences of interrupts.

Fig. 7 shows the interrupts patterns for 6 apps during loading (including e-banking, anti-virus, system pre-installed apps, etc.). The x-axis is the time sequence with 50ms as the interval. The y-axis is the aggregated amount of interrupts observed during the 50ms interval. Their patterns can be told apart even just through human eyes. This phenomenon motivates us to model the UI refreshing process through interrupt time series and detect the foreground app.

C. Adversary Model

We assume the adversary here has tricked the victim to install the malicious app targeting secret unlock pattern or app UI information. The malicious app requires "zero permission" from system for the inference attacks, as reading interrupt statistics is unfettered. Such app is difficult to be detected by mobile anti-virus software or the user during installation time. In addition, our later evaluation on performance shows battery and time consumption is negligible. Thus, it is also hard to be observed by the user at runtime.

For the first attack, either the raw data of interrupts or the inferred unlock pattern is sent out to the remote server of adversary, based on where the analysis happens. Normally, this requires the `INTERNET` permission to be granted. But as discovered by previous works [21], certain covert channels can be exploited (e.g., URI loading by browser) and the permission requirement can be ignored. For the second attack, the UI information can be used locally for subsequent attacks, like phishing.

IV. Unlock Pattern Inference Attack

In this section, we present the attack inferring user's Android unlock pattern. We first elaborate the internal mechanisms of event processing on touchscreen. Then, we introduce our approach in unlock pattern modeling and data processing. In the end, we evaluate the effectiveness and performance of our approach.

A. Touchscreen Controller and Interrupt

A set of mechanisms have been implemented in Android system and the underlying devices to support the process from electrical signal generation from user's finger touching to event dispatching to the receiving app. The complete touch event processing flow is introduced in Appendix VIII-A, and here we only describe mechanism regarding interrupt.

IRQ from Touchscreen. Currently, most touchscreens use the capacitive touch techniques to detect the change of electrostatic field from human's finger in order to capture its movement. Typically, capacitive touchscreens consist of glass as the insulator. The human body is also an electrical conductor, so when the human body comes into contact with the touchscreen, its electrostatic field becomes distorted [22]. When the finger keeps stationary on the screen, the electrostatic field stays unchanged. An IRQ is triggered by the touchscreen hardware when a change happens on the electrostatic field. Therefore, a finger touching or leaving the screen will both trigger one IRQ [23]. When user's finger swipes upon the touchscreen, a sequence of IRQ sequence will be produced as the electrostatic field keeps changing (consider the movement as continuous touching and leaving screen). The amount and frequency of IRQ fired depend on the distance and speed of finger's movement. When the finger moves faster, more IRQs will be generated, but the frequency can not exceed the processing capability (about 135 Hz on Google Nexus 6). Though some kinds of IRQs can be ignored by CPU, this never happens to touchscreen due to its high priority in the processing queue.

B. Attack Methodology

The correlation between finger's movement on touchscreen and IRQ inspires us to infer the unlock pattern through monitoring the interrupt counter. As described in Section III-B, an unlock pattern is composed of a set of swipe lines. Usually, more interrupts can be observed from long swipe lines than short ones. The finger usually pauses at the joint point between two swipe lines, leading to a gap of interrupts. Therefore, by analyzing interrupt data stream, at least partial information on swipe lines (i.e., number and length) can be inferred. Although the exact password pattern is not recovered, the search space is significantly reduced, and enumerating the possible combinations only takes dozens of attempts and minutes of unlocking and waiting time, as shown in our later analysis.

We divide the attack method into the following stages. 1) The public interrupt log file is regularly sampled, and the stream of interrupt counts are preprocessed and divided into *grams* (a segment of ever-changing interrupt counts). 2) We model the unlock pattern into the transition of *states* (a state is a cluster of the swipe lines with the same length) with probabilities. 3) The candidate combinations of states ranked with probabilities are produced for a testing unlock pattern.

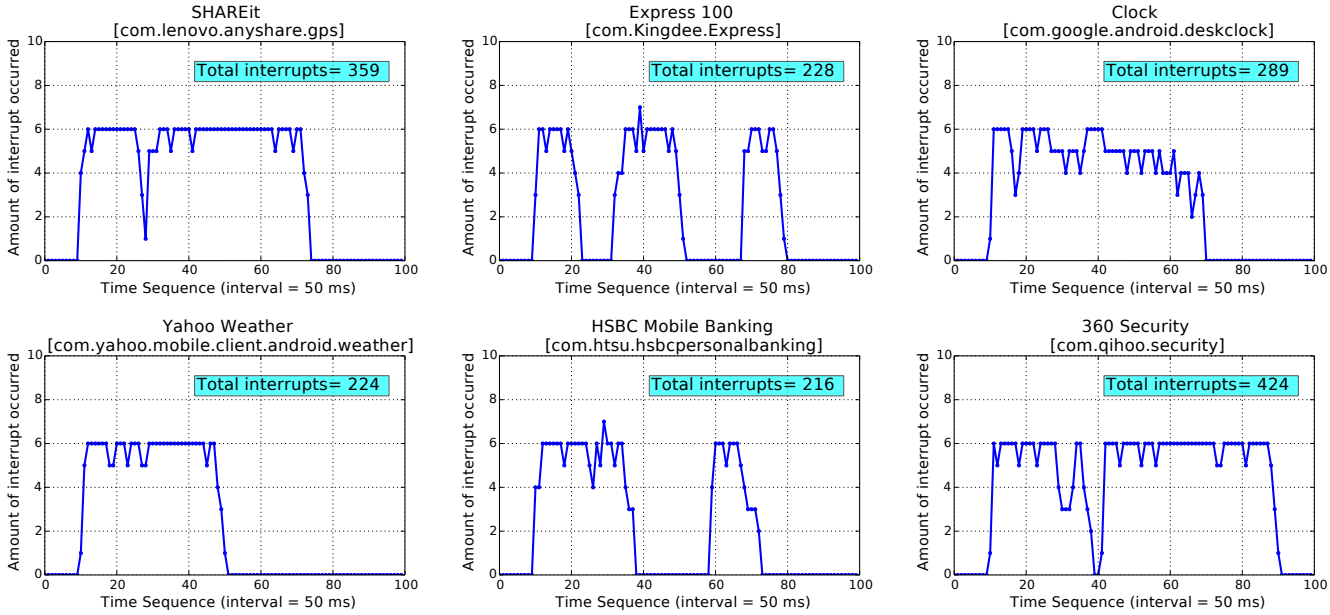

Fig. 7. Interrupt time series of 6 apps' launching processes. The number of interrupts is aggregated in 50ms interval.

TABLE I
SUMMARY OF DEVICE NAMES FOR TOUCHSCREEN CONTROLLER

Phone Model	PIC Name	Device Name	Device Vendor
Google Nexus 6	msmgpio	atmel_mxt_ts	Ateml
Moto Droid Turbo	msmgpio	atmel_mxt_ts	Ateml
Moto Milestone 2	GPIO	qtouch_ts_int	Quantum
Sony Xperia Z3	msmgpio	clearpad	Synaptics
Sony Xperia ion	msmgpio	clearpad	Synaptics
Samsung Galaxy A5	msm_tlmm _v4_irq	mms300-ts	Melfas
Samsung Galaxy S3	s5p_gpioint	melfas-ts	Melfas
Samsung Galaxy S Advance	Nomadik-GPIO	mxt224_ts	Ateml

We model and test the unlock pattern using Google Nexus 6 with AOSP Android 5.1.1 only, but the attack method also applies to other smartphones.

Reading Interrupt Count. The interrupt time series for touchscreen controller could be captured through monitoring `/proc/interrupts`. The first issue we need to address is to find the right entry regarding touchscreen interrupts from the log file. Searching by IRQ number is not a viable solution, as the IRQ number is customized by Android version or manufacturer. Instead, we use device name to identify the entry as it is fixed for the same phone model (`atmel_mxt_ts` for Google Nexus 6). In addition, we surveyed a number of phones and found the device name for touchscreen either contains substring "ts" or "pad" uniquely, as summarized in Table I. Therefore, we can use the substring pattern to find the touchscreen log entry on new phone models. The aggregated interrupt amount since the phone is booted can be read from the column `CPU0`, as shown in Fig. 2, and we sample it at a regular interval.

The sampling frequency of our implementation could reach 1675 Hz, which is much higher than the maximum frequency of touchscreen IRQs (135 Hz on Google Nexus 6) to minimize the odds of missing interrupt updates. In fact, we optimize the data collection stage by only monitoring the interrupts during the unlocking operation. The malicious app we built can stay in the sleeping mode in the background and be activated only when the screen is turned on, which can be detected by registering Android broadcast channel `ACTION_SCREEN_ON`. When the broadcast `ACTION_USER_PRESENT` is observed, the unlocking operation is supposed to be finished, and the app can turn itself back into the sleeping mode.

Data Pre-processing. The sampled interrupt data stream needs to be preprocessed before feeding to the subsequent stages for unlock pattern inference. The steps are elaborated below:

- *Data Deduplication.* We split the interrupt data stream by 1 ms interval. Since our sampling rate (1675 Hz) is higher than 1000 Hz (1 sample per 1 ms), multiple samples could be collected within 1 ms interval. For such case, we keep the first data point within the 1ms interval.
- *Data Interpolation.* Although we use a high sampling rate, occasionally, some changes of interrupts are still missed, especially when heavy computing tasks are run by CPU (around 1.8 % 1-ms intervals have no sample data as observed from the experiments). To fill the missing data points, we use the linear interpolation [24] method, which has been extensively used in the area of signal processing.
- *Interrupt Increment Computation.* The interrupt data obtained are the aggregated values since the bootstrap of the phone. We compute the difference of interrupts counts to get the increment value.

TABLE II
5 STATES OF SWIPE LINES

State	Length	Swipe Line Examples	% of appearance in all state sequences
L_1	1	1→2, 1→4	33.81 %
L_2	$\sqrt{2}$	1→5, 2→4	24.45 %
L_3	2	1→2→3, 3→6→9	10.92 %
L_4	$\sqrt{5}$	1→6, 6→7	26.64 %
L_5	$2\sqrt{2}$	1→5→9, 3→5→7	4.18 %

- *Gram Segmentation.* We segment a long interrupt time series to several grams through searching interrupt gap which is actually the turning points between swipe lines. According to our experiments and empirical analysis, if the amount of interrupts in 60 ms is less than 5, we consider it as a gap. Then we label the gram using the accumulated interrupts increments within the time window. For example, the interrupt time series in Fig. 5 is converted to 4-grams {28, 58, 77, 45}.

Unlock Pattern Modeling. A gram is labeled by the number of observed interrupts, and we need to find the mapping from it to the corresponding swipe lines. Since the length of a swipe line is proportional to the number of interrupts, we cluster the swipe lines by the length and the grams by the interrupt count and build the correlation. It turns out there are 5 types of swipe lines, associated with 5 types of grams.

For explanation, we model the unlock pattern in 2-dimensional Cartesian coordinate system and set dot 7 as origin point with coordinate as $(0, 0)$, as shown in Fig. 4. Therefore, the swipe line from dot 7 $(0, 0)$ to dot 8 $(0, 1)$ is represented by a vector $[0, 1]$ with length 1. Under this model, all swipe lines can be clustered into 5 categories (or *states*), labeled as L_i, $i \in [1, 5]$, which are listed in Table II. Based on such model, every unlock pattern could be represented by a *state sequence*. For example, the pattern `41235789` could be translated to $L_1L_3L_5L_3$. A long swipe line can be represented by one or two states (e.g., `147` corresponds to L_1L_1 or L_3). To make sure that the mapping from swipe line to state is unique, we always use the state with the longest length (so `147` is mapped to L_3). On the contrary, the mapping from state to swipe line is one-to-many (see Table II). The number of unlock patterns associated with one state sequence is yet limited, due to the restrictions of valid unlock pattern. On average, one state sequence corresponds to 20.37 patterns.

Single State Analysis. Our goal is to infer the state sequence from the grams. As the first step, we need to derive the correct state from a single gram. This task looks trivial at first sight: one simple approach is to correlate the state with the range of interrupt count and classify a gram into a state if it falls within the range. Unfortunately, this approach easily failed due to the big variance of people's drawing actions. Even for a single user, the way of swiping a line is different from time to time. This forces us to find a model which can handle the variances, instead of a simple linear equation.

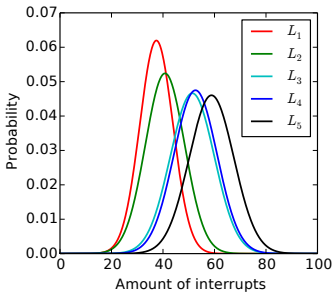

Fig. 9. Gaussian distributions of 5 states.

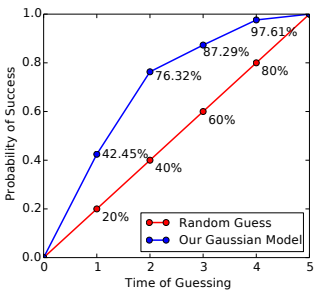

Fig. 10. Probability of successful guess. 5 attempts are needed at most.

To uncover the relationship, we carried out a user study and recruited 5 users to join our experiments[1]. We asked each user to draw lines belonging to each state for 100 times on the unlock pattern UI of Google Nexus 6. At the same time, a self-developed app runs in the background and samples the interrupt count. Fig. 8 shows the interrupt accumulation histogram of these 5 states.

As shown in the histograms, the amount of interrupts across different swipes forms Gaussian-like unimodal distribution (normal distribution), for all 5 states. Such observation inspires us to compute the probabilities of 5 states derived from a gram, using the Gaussian model trained from real-user data. In particular, we use probability density function (PDF):

$$f(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{(x-\mu)^2}{2\sigma^2}) \tag{1}$$

In our case, $f(x|\mu, \sigma)$ is the probability of state, x is the amount of interrupts, σ is the expectation, and μ is the standard deviation. Leveraging the curve fitting functionality of MATLAB [25], σ and μ could be obtained from the same set of data collected from the 5 volunteers (also shown in Fig. 8). Given an interrupt amount x, we calculate probabilities of states L_i, $i \in [1, 5]$ it belongs to as:

$$\Pr(y|L_i) = \begin{cases} f(y|37.43, 6.439), & i = 1 \\ f(y|40.79, 7.611), & i = 2 \\ f(y|51.38, 8.555), & i = 3 \\ f(y|52.62, 8.399), & i = 4 \\ f(y|58.84, 8.665), & i = 5 \end{cases} \tag{2}$$

It turns out the distribution of the 5 states can be separated in most cases, as plotted in Fig. 9. The only exception is for L_3 and L_4. This can be explained by their length. In fact, the geometric length of L_3 and L_4 is 2 and $\sqrt{5}$, and the difference is only 0.23, much less than other pairs.

In order to evaluate the accuracy of our Gaussian model, we designed a simulation experiment. According to the percentages of every state (listed in Table II) and distribution features (shown in Fig. 8), we generated 1,000,000 simulated interrupt amount observation values with state label, (y, L_i). Based on these simulated data, the probability of correct guessing for

[1]We have got the IRB approval from the authors' institutes before performing any experiment related to human subjects.

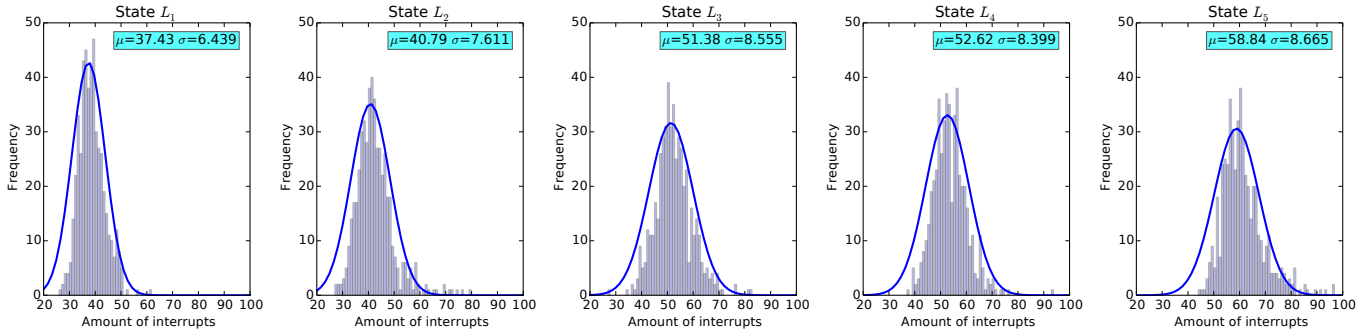

Fig. 8. The distribution of interrupt amounts for 5 states.

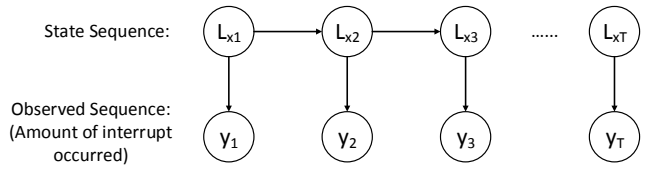

Fig. 11. Using HMM to infer state sequence. $\{L_{x1}L_{x2}L_{x3}\dots L_{xT}\}$ can be inferred using $\{y_1y_2y_3\dots y_T\}$.

1 to 5 times using our Gaussian model is calculated and the result is shown in Fig. 10. The success rate is substantially increased: even for one-time guessing, the success rate is 42.45 %, doubled from that of random model.

State Sequence Analysis. After the probability of a single state is computed, the next step is to derive the state sequence. Within one unlock pattern, states are not independent and the probability of one state is affected by previous ones. For instance, when a user swipes a line of L_5 (1→5→9 or 3→5→7), she can not swipe L_5 for next. Therefore, we treat the problem of finding state sequence as a process of solving Hidden Markov Model (HMM) [26], that is to find the correct (hidden) state sequence from the observed sequence (grams), as shown in Fig. 11.

Viterbi Algorithm. Viterbi algorithm [27], [28] is a classic recursive optimal solution for searching the most likely sequence of hidden states, which is particularly suited for solving HMM. We formalize this algorithm for our settings: suppose the state space of HMM is $S = L_i, i \in [1,5]$, the probability of initial state L_i is π_{L_i} and the transition probability from L_i to L_j is Tr_{L_i,L_j}. If the observation sequence is $\{y_1, y_2, \dots, y_T\}$ (every element is the interrupt amount of a gram), the most likely state sequence $\{L_{x1}, L_{x2}, \dots, L_{xT}\}$ could be calculated through:

$$V_{1,L_k} = \Pr(y_1|L_k) \cdot \pi_{L_k} \tag{3}$$

$$V_{t,L_k} = \max_{L_x \in S}\{V_{t-1,L_x} \cdot Tr_{L_x,L_k} \cdot \Pr(y_t|L_k)\} \tag{4}$$

where V_{t,L_k} is the probability of the most likely state sequence with the first t observations and L_k as the final state. $\Pr(y_t|L_k)$ is the emission probability of showing observation y_t in the hidden state L_k. So, to apply Viterbi algorithm, we need a way to represent emission probability $\Pr(y_t|L_i), i \in [1,5]$ and transition probability $Tr_{L_i,L_j}, i,j \in [1,5]$.

Emission Probability. Since the probability for each state L_i for a gram has been calculated in the previous step using Equation 2, we assign the emission probability $\Pr(y_t|L_i)$ with these values.

Transition Probability. We build set M containing the mappings between all the 389,112 pattern passwords and state sequences (e.g., [41235789⟶$L_1L_3L_5L_3$]), and use it to infer the probability per state sequence. Different from the standard Viterbi algorithm, the transition probability at step t in our case does not only rely on the one state ahead, but all previously encountered states (the sequence of previous states is defined as $L_{seq(t-1)}$). Thus, we customize the formula of transition probability as below:

$$Tr_{L_{seq(t-1)},L_j} = \frac{\sum M^{[s]}.start\_with(L_{seq(t-1)\|L_j})}{\sum M^{[s]}.start\_with(L_{seq(t-1)})} \tag{5}$$

where $\sum M^{[s]}.start\_with(L_{seq(t-1)})$ is the amount of state sequences that start with $seq(t-1)$ and contain s states (the target unlock pattern is s-gram, which is determined during the previous data pre-processing stage).

Top-N Result. The output of Viterbi algorithm is the most likely state sequence, which may be incorrect sometimes. Therefore, we could provide N most likely state sequences ranked by the overall probabilities to increase the chances of successful attacks. The algorithm is shown in Algorithm 1.

Algorithm 1: Top N state sequences calculation

```
for i=1 to N do
    StateSeq = Viterbi(ObservedSeq);
    result.add(StateSeq) ;   // record Top X result
    M.remove(StateSeq) ; // adjust M to change Tr
end
output result;
```

Pattern Password Recovery. A state sequence inferred in the last step could be shared by multiple pattern passwords (e.g., both 1235789 and 7415963 can be described by $L_3L_5L_3$). The concrete pattern password could be obtained by attempting the combinations of digits corresponding to the state

TABLE III
SEARCH SPACE AFTER SUCCESSFUL STATE SEQUENCE INFERENCE

Pattern	2-gram	3-gram	4-gram	5-gram
# of Patterns	12.9	25.7	29.2	29.9
Space Reduction	99.997 %	99.993 %	99.992 %	99.992 %

sequence. It turns out the search space is significantly reduced when starting from state sequences, as shown in Table III. If the phone is grabbed by an attacker, only dozens of attempts are needed for a correctly interred state sequence. Take 3-gram pattern as an example, 99.993 % patterns have been excluded already and on average only 25.7 pattern passwords need to be tested. If we assume drawing one pattern takes 4 s, an attacker could unlock victim's phone in $\lceil 25.7 \rceil \times 4 + 4 \times 30 = 224$ s (every 5 wrong attempts lead to 30 s punitive wait, which is the default setting in AOSP). This process can even be fully automated by plugging in a signal simulator to the victim's phone [29].

C. Evaluation

We evaluate the effectiveness of our attack against the pattern passwords inputted by real users. Different from previous works [2], [30], which only attack a very limited number of patterns (50 for [2] and 1 for [30]), our attack targets **all 389,112** patterns. In particular, we first evaluate the accuracy of gram segmentation during the data pre-processing stage. Then, we examine the success rate of the state sequence inference. As a comparison, we also run the attack under the same setting of [2].

Attack App. Two modules, interrupt sampling module and data analysis module, are developed and included in the attack app. For the first module, we wrote it in native C using Android NDK [31]. The second module is written in Java. For the optimal performance, we implemented the HMM and Viterbi algorithm (around 750 lines of code) instead of using other general libraries.

Experimental Setup. The Gaussian model for single state needs to be trained before the actual attack, and we reuse the data collected from the 5 users. For testing, we invited 2 users and none of them participated in the training step. The experiment device is the same Google Nexus 6 phone with our attack app installed.

We only consider 2-gram, 3-gram, 4-gram and 5-gram patterns, because too long gram patterns are rarely used in practice [17], [32]. Besides, it is difficult to require the users during our test to remember all long pattern passwords. We randomly generated 20 password patterns for each x-gram ($x \in [2, 5]$) from 389,112 pattern passwords (listed in the Appendix VIII-D) and asked these two users to draw each generated pattern two times. In total, we obtained 160 password patterns from each user.

Gram Segmentation Result. In this part, we examine whether the interrupt time series could be segmented correctly and the result is shown in Table IV. The success rate turns out to be very high (more than 95 % on average), which also suggests the interrupt gap between different swipe lines is prominent. From the perspective of computation complexity, even if the attacker's knowledge is only the number of grams, the search space can be substantially reduced.

TABLE IV
SUCCESS RATE FOR GRAM SEGMENTING

Pattern	Success Rate	Search Space Reduction
2-gram	98.75 %	99.96 % (from 389,112 to 168)
3-gram	92.5 %	99.35 % (from 389,112 to 2,544)
4-gram	97.5 %	97.16 % (from 389,112 to 11,048)
5-gram	97.5 %	90.45 % (from 389,112 to 37,160)

TABLE V
SUCCESS RATE FOR STATE SEQUENCE INFERENCE

User #	Top N	2-gram	3-gram	4-gram	5-gram	Popular
User 1	Top 3	50 %	25 %	7.5 %	0	47.2 %
	Top 5	80 %	27.5 %	10 %	0	52.8 %
	Top 10	97.5 %	40 %	20 %	2.5 %	61.1 %
	Top 20	97.5 %	60 %	37.5 %	12.5 %	72.2 %
	Top 40	97.5 %	90 %	52.5 %	17.5 %	83.3 %
User 2	Top 3	45 %	20 %	15 %	2.5 %	50 %
	Top 5	62.4 %	22.5 %	22.5 %	5 %	61.1 %
	Top 10	95 %	35 %	25 %	10 %	63.9 %
	Top 20	100 %	50 %	40 %	20 %	75 %
	Top 40	100 %	70 %	57.5 %	22.5 %	77.8 %

State Sequence Inference Result. We tested the effectiveness of the output of state sequence inference and the result is listed in Table V. In the case of 3-gram, random guessing only reaches 0.0157 % success rate (guessing 3 times) while our attack could improve the success rate to thousands of times – 20 % at least. Given that exhausting passwords for a 3-gram pattern is only 25.7 times (see Table III), for 20% such patterns, the attack time is acceptable. The success rate decreases with the increase of the number of grams since more errors would be introduced.

Popular Patterns. Recent studies [17], [32], [33] on usable security discovered that the pattern used by a user is not a random selection. In fact, several patterns are extensively used, and we could leverage this insight to remove unpopular patterns from search space. We studied the 6 popular patterns [32] (ranging from 2-gram to 5-gram) and removed any pattern from the 389,112 patterns if it contains a L_4 line or starting dot is {5, 6, 8, 9}. L_4 line is hard to be drawn by user, and none of the starting dot is used by popular patterns.

To test our attack against popular patterns, we use the 6 patterns as the initial set and extend it through clock-wise rotations (90, 180 degrees separately), totally 18 patterns. The shapes of the original 6 patterns and the list of all 18 patterns are shown in Appendix VIII-D. We asked the same 2 users to input the 18 patterns twice. The result shows our success rate is improved noticeably (see the last column in Table V).

Password Pattern Inference with Prior Knowledge. The previous experiments consider all valid password patterns

TABLE VI
SUCCESSFUL RATE FOR PATTERN PASSWORD INFERENCE

User #	Top N	50 patterns	100 patterns	200 patterns	500 patterns
User 1	Top 1	38 %	29 %	23.8 %	11.5 %
	Top 2	56.5 %	40.3 %	28.5 %	16.7 %
	Top 3	60 %	46.3 %	29.5 %	18.7 %
	Top 5	60 %	49.7 %	32.3 %	20.4 %
User 2	Top 1	38.5 %	31.1 %	22.3 %	12.2 %
	Top 2	61.1 %	42 %	27.5 %	17 %
	Top 3	64 %	47.1 %	29 %	18.6 %
	Top 5	64 %	51.5 %	32.3 %	19.6 %

as targets. For this experiment, we adopt the same setting of previous works [2], that user's password choices can be confined based on the prior knowledge. We assume the victim selects her secret pattern passwords from a pre-defined set. Also in this experiment, we evaluate the success rate of breaking the password pattern instead of state sequence.

We asked the two users each to select one pattern from the 80 patterns (4×20) provided in the previous experimental setup and draw it on the testing phone twice. This process was repeated 20 times. The selected pattern is then mixed with randomly generated password patterns to build the pre-defined set. Four different set sizes are considered here: 50, 100, 200 and 500 and the password inference success rate is listed in Table VI. When the set size is 50, we have more than half a chance to unlock victim's phone by just 2 attempts. This result is comparable to [2] exploiting the side-channel from accelerometer, which was able to crack a password pattern within 5 attempts with 73 % accuracy when user's sitting and 40 % accuracy when user's walking. In addition, our attack removes the two restrictions of [2]: 1) the body movement has to be small (the accuracy is much worse when the user walks than sits). 2) the mobile phone has to be held at hand (accelerometer produces no usable data when the phone is placed on the desk).

Battery and Time Consumption. Most of the battery consumption is cost by the interrupt sampling module. Since this module only runs when the screen is lighted and ends before the phone is unlocked (generally, the sampling period $<30\ s$), the battery consumption is very slim and hard to be observed (<1 %). To infer one unlock pattern, the computation time of data analysis module is less than 0.3 s, which is also negligible.

V. FOREGROUND APP INFERENCE ATTACK

In this section, we present the attack on inferring the app running in the foreground. We start from introducing Display Sub-System and interrupts. We then elaborate how we leverage the interrupt side-channel for attacks and the evaluation result.

A. Display Sub-System and Interrupt

Display Sub-System (DSS) takes in charge of controlling the actual display and governing the FrameBuffer driver. It keeps refreshing the screen using the content from FrameBuffer (`/dev/graphics/fbX`) when the content is updated till all the changes are rendered. The complete workflow of Android display system is provided in Appendix VIII-B as supplementary.

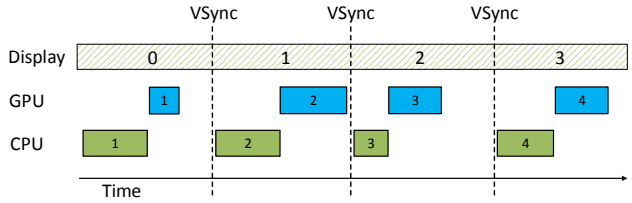

Fig. 12. VSync signal that keeps refreshing rate fixed.

IRQ from DSS. The design of screen refreshing is changed drastically since Android 4.1, and our attack targets the DSS under this setting. In Android 4.1, Project Butter is launched by Google to improve UI display smoothness. As one of the main visual performance improvements, the VSync (vertical synchronization) mechanism is integrated [34] to keep the refresh rate fixed at 60 Hz, or 60 frames per second (FPS). Specifically, the refresh requests will be queued and synchronized at regular interval. Fig. 12 illustrates this VSync mechanism, in which the drawing operations of CPU and GPU always start with the VSync signal [35].

A VSync IRQ will be issued by DSS after one full-screen refresh is completed [36], announcing the end of one frame interval and the beginning of the next. As shown in Section III-B, the IRQ time series is distinctive for each app's launching process, depending on the content loaded by the app and its refreshing strategy. Since only one app is allowed to run in the foreground, we could infer the foreground app through observed interrupt time series from DSS.

Remarks. For some phones under VSync mechanism, the frequency of interrupts issued from DSS can be 120 Hz, due to VSync signal virtualization [37], a new enhancement by Google since Android 4.4. This technique is proposed for more efficient synchronization. Two virtual VSync signals (one is used for app UI data preparation, and another is for SurfaceFlinger) will be sprung from one physical VSync signal.

B. Attack Methodology

Different from the unlock pattern inference attack, where the interrupt time series can be segmented and the amount of interrupts within each segment can be mapped to a limited set of states, the interrupt time series of app launching is more random and it is impractical to map the foreground app to an arbitrary one. Therefore, we build *app fingerprints* using interrupt time series for popular apps a priori and attempt to find a matching app for a foreground app running on victim's phone. We first elaborate our techniques for reading interrupt data, data pre-processing and similarity calculation. Then we describe the training process for building fingerprint base and testing process for detecting foreground app.

TABLE VII
SUMMARY OF DEVICE NAMES FOR DSS

Phone Model	PIC Name	Device Name	Device Vendor
Google Nexus 6	GIC	MDSS	Qualcomm
Moto Droid Turbo	GIC	MDSS	Qualcomm
Moto Milestone 2	INTC	OMAP DSS	TI
Sony Xperia Z3	GIC	MDSS	Qualcomm
Sony Xperia ion	GIC	MDP	Qualcomm
Samsung Galaxy A5	GIC	MDSS	Qualcomm
Samsung Galaxy S3	COMBINER	s3cfb	Samsung
Samsung Galaxy S Advance	GIC	nmk-i2c (non-unique)	ST-Ericsson

The training and testing are also done on Google Nexus 6. We believe our techniques could be applied without modification on other devices with Android 4.1 above installed.

Reading Interrupt Count. The interrupt time series for DSS could be captured through monitoring `/proc/interrupts` as well. Following the similar method as Section IV-B, we use the device name to identify the entry for DSS. We look for `MDSS`[2] on Google Nexus 6. The device names of DSS on several other phones are summarized in Table VII. All interrupts for DSS are logged under `CPU0`, so we only need to monitor that column.

The sampling frequency of our implementation could reach 4899 Hz[3], which is much higher than the frequency of IRQ from DSS (60 Hz or 120 Hz). High sampling rate would lead to more power consumption, but as described in Section V-C), the performance impact to the phone is still limited. For power saving, we could instruct the malicious app to run only when the screen is turned on (registering `ACTION_SCREEN_ON`) and to sleep when the screen is locked (registering `ACTION_SCREEN_OFF`). The starting point of sampling (i.e., app launching) could be determined through combing the knowledge of targeted app and our previous interrupt channel for touchscreen in Section IV. For instance, after the malicious app finds the user presses the Home Key (through the system broadcast `ACTION_CLOSE_SYSTEM_DIALOGS`) and two successive interrupts for touchscreen are detected, it can learn that the user clicks an app icon on the home screen. The sampling period could be set according to the launching time duration of the targeted apps.

Data Pre-processing. Similar to the unlock pattern inference attack, deduplication and interpolation are applied in the same way (see Section IV-B). An additional noise filtering step is introduced to removed the background noise on interrupts. The interrupts from DSS are not only caused by the app running in the foreground. The system UI events (status bar showing time, signal strength, battery, etc.) also cause screen refreshing, incurring noises. By inspecting the interrupt sequence caused by system UI events, we found all the noises are segments of consecutive 50 ms intervals with less than 30 interrupts in total and less than 6 interrupts per 50 ms interval. They are nominal comparing to the interrupts from the foreground app, and we search the interrupt time series to remove all such noises.

[2]The name of `MDSS` is used by Qualcomm CPU series and stands for Mobile Display Sub-System.

[3]Compared with the interrupt sampling module for unlock pattern inference, the reason for different frequencies stems from the API `fgets` which is used to read `/proc/interrupts`. The sampling rate is affected by the location of interrupt log entry in the file.

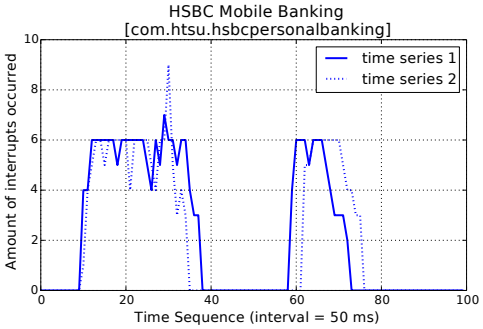

Fig. 13. The curves of interrupt time series are similar for the same app.

App Similarity Calculation. We consider the interrupt time series after pre-processing as app's fingerprint, which is used to determine whether two apps are the same one. Exact matching is not a viable solution here. Even for the same app, the screen refreshing during app launching is not the same, because of the background processes and different network connection status. We use sequence similarity as the metric to adapt to such unstable situations. Fig. 13 shows an example in which we could find, for the two interrupt time series coming from the same app, their curves are quite similar, but not coincide at every timestamp.

For calculating the sequence similarity, several difficulties must be overcome: the lengths of two interrupt time series may be different, and there may exist displacements along the timeline. After examining different matching algorithms, we found Dynamic Time Warping (DTW) algorithm [38] achieves the optimal result. DTW is designed to calculate intuitive distances between time series by ignoring shifts in the time dimension. Its basic idea is to find a minimum-distance warp path of which length is treated as the measured distance. We formalize it like [39] in our settings: given two interrupt time series

$$X = \{x_1, x_2, \ldots, x_i, \ldots, x_{|X|}\}$$

and

$$Y = \{y_1, y_2, \ldots, y_i, \ldots, y_{|Y|}\}$$

, a warp path

$$W = \{w_1, w_2, \ldots, w_K\}, \quad \max(|X|, |Y|) \leq K \leq |X| + |Y|$$

is constructed where K is the length of the warp path and the k-th element of the warp path is $w_k = (i, j)$ where i is an index from X and j is an index from Y.

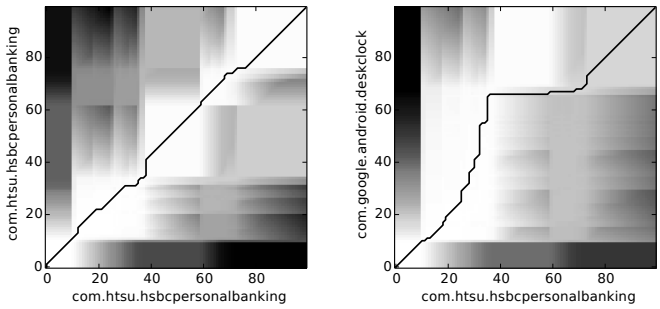

Fig. 14. Warp paths of time series. Left: the same app. Right: different apps.

The optimal warp path is the one leading to minimum warp distance, where the distance of a warp path W can be defined as

$$Dist(W) = \sum_{k=1}^{k=K} Dist(w_{ki}, w_{kj}) \quad (6)$$

where $Dist(w_{ki}, w_{kj})$ is the distance between the two data point indexes in the k-th element of the warp path.

The DTW distance could be used as the measurement of the similarity between two interrupt time series. Fig. 14 gives two warp path examples in which the background is the cost matrices of distances (the deeper gray, the higher cost and the farther distance). The warp path based on the same app (the same two series as Fig. 13) in the left figure is nearly a straight line from $(0, 0)$ to $(100, 100)$. However, the path in the right figure based on different apps has several significant turnings.

Training Phase. In this phase, we profile apps of our interest and build the fingerprint database. We automate the training process by using monkeyrunner [40] which can repeatedly open and close apps following the scripted instructions. While deciding when to open an app is easy (the time after the exit of the prior app), it is not clear when to close the app. Based on empirical observations and experiments, we found 4.5 s is enough for an app to finish loading. Therefore, monkeyrunner will close the app after 4.5 s since starting. For every app, monkeyrunner triggers the launching operations for N times, and the derived N fingerprints are all stored in the fingerprint database. N can be adjusted by the adversary. While a small N could lead to mismatch for the same app, big N will incur large overhead in comparing. We set N to 10 by default.

Testing Phase. This phase is to test whether an app running in the foreground matches one in the training dataset. The two steps included in this phase are described below:

- *Pre-filtering.* Computing DTW distance is costly. To reduce the overhead, we use the total amount of interrupts as a condition for pre-filtering. Specifically, we obtain the upper-bound and lower-bound of the total amount of interrupts for all apps in the training dataset. If the amount falls out of the range from training dataset, which is extended from the upper- and lower-bound by 25 % (see Section V-C for how 25 is decided), the app is considered irrelevant and not proceeded to the next stage.
- *Classification.* DTW distance is supposed to be calculated between the fingerprint of the testing app and all fingerprints in the training set. For optimization, we apply the same heuristic in pre-filtering stage and skip the distance calculation if the interrupts total amount greatly differs. When DTW distance needs to be calculated, we employ FastDTW algorithm [39] to accelerate the computation. After the fingerprint distances between the testing app and training apps are computed, we use k-nearest neighbors (k-NN) algorithm [41] to classify the app, that is a majority vote by its neighbors. For instance, assuming $k = 5$, if the testing fingerprint matches 3 fingerprints from app_a and 2 fingerprints from app_b, we consider app_a is running in the foreground. Since the result of majority vote may be incorrect and several fingerprints in the training set may have the same distances as the testing fingerprint, we also consider the top-N results. In addition, to avoid identifying an app not in the training set, we could customize the training set based on the list of installed apps on the victim's phone. Such list could be easily obtained by invoking `PackageManager` and `PackageInfo` classes without permission.

C. Evaluation

We evaluate the effectiveness of our attack using interrupt data collected by running popular apps. In addition, we measure the statistics of the interrupt amount per app and justify how the pre-filtering threshold is determined. The performance overhead is tested using variant sampling rate and in the end, we show an advanced version of this attack in sniffing the foreground Activity in Appendix VIII-C.

Attack App. The attack app contains two modules – interrupt sampling module (built with Android NDK) and data analysis module (about 700 Java lines of code). Our implementation of DTW distance calculation is based on Java-ML library [42].

Experimental Setup. We select 100 popular apps from Google Play to build the training set, as listed in the Appendix VIII-E. These apps all stay in the foreground when launched, and the apps always running in the background, like instant messaging apps, are not included. Each app is launched 10 times, and 1,000 fingerprints are recorded in total. For some apps, an introduction or user agreement page is displayed at the first launching after installation and never shown afterward. The fingerprints, in this case, are discarded by us manually. To build the testing set, we randomly select 10 apps (as shown in Table IX) from these 100 apps in the training set, run each one 10 times, and record 100 fingerprints in total.

Interrupt Amount Threshold. We use the range of interrupt amount to pre-filter apps and optimize similarity calculation. The threshold θ for separating fingerprints needs to be determined before testing. For this purpose, we look into the distribution of interrupt amount across different apps and within one app. For every app in the training set, we count the mean interrupt amount of its 10 fingerprints, as shown in Fig. 15 in ascending order. The maximum value is 635.7

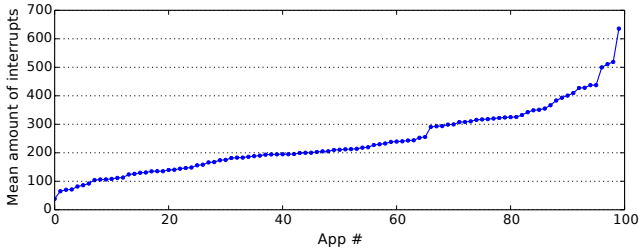

Fig. 15. Mean interrupt amount.

TABLE VIII
SUCCESS RATE OF APP IDENTIFICATION UNDER DIFFERENT K

k	k=3	k=5	k=7	k=9
Top 1	77 %	87 %	83 %	82 %
Top 2	85 %	91 %	88 %	90 %
Top 5	93 %	95 %	94 %	93 %
Top 10	94 %	96 %	96 %	98 %

coming from Microsoft Hyperlapse Mobile: during launching, its UI is always refreshing and even the background of main Activity is dynamic. The minimal value is 38.5 coming from Google Search App: its launching process is very fast and no animation is used, which is reasonable for a search engine app. The average and standard deviation of interrupt amount are 237.3 and 114 respectively.

For a single app, the interrupt amount has much less variation. Among the 10 samples of every app in the training set, the average fluctuation is 20.8 %. Thus, we set the threshold θ to 25 % to accommodate some redundancy.

App Inference Result. We apply k-NN algorithm to classify a testing app, and the selection of k affects the classification precision. Table VIII lists the result under different k. When k = 5, the success rate is the highest. Even for a one-time test, on average, there is 87 % chance for the adversary to know which app runs in the foreground. Also, we can perfectly identify some tested apps, such as `com.cleanmaster.mguard` (Table IX).

We notice the launching patterns of some apps with similar names can not be distinguished, like:

`com.google.android.apps.docs.editors.docs`

TABLE IX
SUCCESS RATE OF APP IDENTIFICATION, K = 5

App Name	Top 1	Top 2	Top 5
tv.danmaku.bili	100 %	100 %	100 %
com.baidu.search	80 %	90 %	90 %
com.icoolme.android.weather	90 %	90 %	90 %
com.scb.breezebanking.hk	80 %	90 %	100 %
ctrip.android.view	50 %	50 %	60 %
com.lenovo.anyshare.gps	100 %	100 %	100 %
com.sometimeswefly.littlealchemy	100 %	100 %	100 %
io.silvrr.silvrrwallet.hk	90 %	100 %	100 %
com.cleanmaster.mguard	100 %	100 %	100 %
com.ted.android	80 %	90 %	100 %

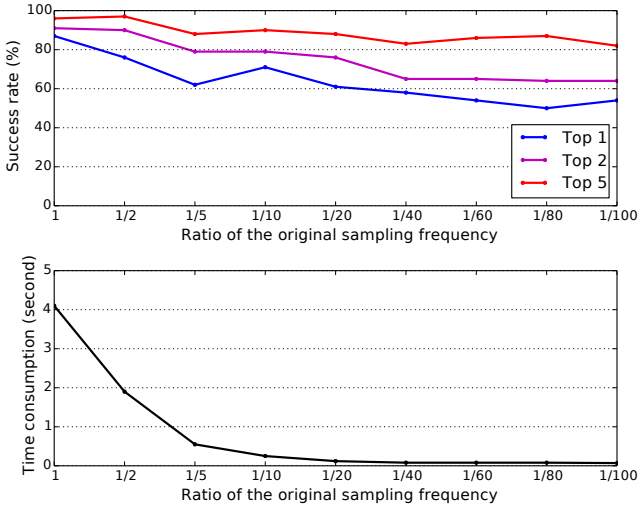

Fig. 16. The result under varying sampling frequency.

`com.google.android.apps.docs.editors.sheets`
`com.google.android.apps.docs.editors.slides`

The main reason for such misclassification is code reusing. Clearly, a vendor or a developer prefers to keep the uniform UI style for a series of apps. The number of such apps, however, are limited among the most popular apps.

Battery and Time Consumption. We mainly consider the battery usage for interrupt sampling module, which reads `/proc/interrupts` periodically. When it is running in the background, 13 % CPU resources will be occupied and 1 % battery is consumed per 6 min.

In the above settings, one time DTW distance calculation costs about 15.8 ms. However, this time consumption is not a stable value and affected by the number of non-zero interrupt count. One complete app identification (classification) costs 4.1 s, which depends on the size of the training set. This value is not a simple multiple of overhead from distance calculation, as the times required for distance calculation depends on the result from the pre-filtering stage.

In order to reduce the battery and time consumption, one solution is to reduce the sampling frequency of interrupt sampling module, which may impact the inference accuracy. To quantify such relationship, we carried out experiments under different sampling frequencies. The result is illustrated in Fig. 16 and the time overhead decreases rapidly without significantly impacting the successful rate. For instance, 1/10 sampling frequency of the default value (4899 Hz) could still guarantee 71 % successfully rate and the time overhead drops to only 0.25 s. Under this configuration, the adversary could sneak the malicious activity into the screen more timely.

Discussion. In the experiments above, the testing phone is not running many background apps. We repeated the experiments under a heavy workload running environment to see if the attack result is stable under different environment, i.e., the available memory is less than 30 % when too many processes are running. Such run-time environment affects the app launch-

ing process and the sampling frequency of our attack app at the same time. The result shows the average identification rate for the 10 testing apps could still reach 72 % for one guess (77 % for top 2 and 84 % for top 5).

VI. Discussion

Our attacks successfully exploit the leaks from touchscreen and display interrupts, while whether other interrupts can be exploited is unclear. We first discuss the potential threats from other interrupts and then suggest several defenses.

A. Leaks from Other Interrupts

A lot of peripheral devices have been introduced to Android, among which a large portion has access to private information, as described in Section II. We believe the attack surface on interrupt is not exhausted, and new attacks may be sprung from other interrupt sources. For example, we can acquire the interrupt information from Bluetooth device (named `bluetooth hostwake`) and NFC Controller device (named `bcm2079x`) in the interrupt logs on Google Nexus 6 to infer when the devices are running or sleeping. Furthermore, the size/timing of the file/packet transmitted through these communication channels can be inferred potentially. These could give the attacker a big lift in information stealing. As an example, since NFC is largely leveraged for in-store payment, an adversary can do targeted phishing to steal user's credit card number when knowing the status of NFC device. What's more, it is also a reasonable speculation that such information leakage exists on other operating system platforms and can be exploited. It is necessary to fully explore the attack surface via interrupts, but the effort is, however, considerable. We believe an approach based on automated testing can greatly reduce the overhead and increase the chances of discovering new interrupt leaks, and we leave it as the next step.

B. Defense

The attacks presented in our work belong to a big category of side-channel attacks. It is known by the community that side-channel attacks are hard to detect and mitigate, due to their stealthy nature. Recent work by Zhang et al. [43] proposed a new detection system against runtime side-channel attacks on Android and also released an app on Google Play. We tested this app against our attacks but found none of our attacks were detected or prevented.

On the other hand, we believe Android needs to be fortified at the system level. Since the interrupt statistics leak from `proc` filesystem (`/proc/interrupts`), the natural idea is to remove the `proc` filesystem or make it invisible to processes. However, these simple remedies would cause big usability and compatibility issues as many utilities rely on `/proc` to gain access to Linux kernel information [44]. For example, the Linux command `ps` relies on the `/proc/<pid>/` to obtain process status [45], and the `irqbalance` service [46] uses interrupt statistics information for balancing CPU load on a multiprocessor system. Instead, we suggest two alternative defenses below:

Fine-grained Access Control on `procfs`. The access to `proc` filesystem should be mediated. Linux has been moving towards this direction and access to some files under `proc` filesystem is restricted. For example, `/proc/vmallocinfo` is not world-readable. Still, not all `proc` files are protected, including the one we identified. In the long run, we believe all files under `proc` filesystem should be scrutinized and protected at different levels (accessed by system process only, accessed based on granted permission, or open to public). Yet, the decision should be made after measuring the impact on legacy apps and OS components to ensure their functionalities are not largely disrupted.

Decreasing the Resolution of Interrupt Data. Similar to the defense proposed by Zhou et al. [21] which foils the attack by rounding up or down the data value from exploitable sources (e.g., the volume of network traffic logged in public statistics), we could reduce the resolution of logged interrupt data as mitigation. Noise injection, proposed by Xiao et al. [47], is also an alternative solution in the same direction.

As one option, `/proc/interrupts` can present the hardware interrupt information grouped by PIC (it connects to multiple devices) instead of a single device. Another option is to update the interrupt count after a number of interrupts have been collected. Since the precision is degraded, different touch movements or UI refreshes may share the same delta of interrupts count, reducing the chance of finding the right unlock pattern or the foreground app by attackers.

VII. Related Work

A. Side-channel Attacks

Linux `procfs`. Zhang and Wang [6] were the first to present a side-channel attack by exploiting Linux `proc` filesystem, which allows a malicious user to eavesdrop other users' keystrokes. By tracking changes in the application's memory footprint (`/proc/<pid>/statm`), Jana et al. [5] showed that how a malicious Android app can infer which page a user is browsing. Zhou et al. [21] demonstrated several attacks to infer user's identity, location using such side channel information (e.g., `/proc/uid_stat/`). UI state can also be read from `proc` filesystem, as described by Chen et al. [18]. Moreover, Lin et al. [48] exploited `/proc/<pid>/stat` to detect target apps activities. The TCP sequence number inference attack of Qian et al. [49] and the ret2dir attacks of Kemerlis et al. [50] exploited `/proc` information as well. Compared with previous works, our work is the first one exploiting `/proc/interrupts` to implement inference attacks and we proposed a new approach for general interrupt timing analysis. In addition, our work makes the first step to investigate the security issues coming from the integration of the emerging hardware components and the legacy kernel on mobile platforms.

Leaks from Sensors. In addition to Linux `procfs`, the reading from sensors can also be exploited by malicious apps for side-channel attacks. As demonstrated by previous works [1], [2], [51], [52], [53], the data stream from the

accelerometer can be leveraged by malicious apps to infer mobile phone user's tapping locations on screens or even password. Michalevsky et al. [54] demonstrated an attack which is able to identify the speaker information and parse the speech through reading phone gyroscopes. More recently, the leaks from the sensors on smartwatch were investigated. Wang et al. [55] demonstrated that user's keypresses on QWERTY keyboard can be inferred using accelerometer and gyroscope data. Liu et al. [56] showed that side-channel information from accelerometer and microphone can be leveraged to infer PIN typed on numeric keypad and text typed on QWERTY keyboard.

B. Timing Analysis

Techniques for timing analysis have been extensively used for inference attacks. Kocher et al. [57] showed that secret keys used by DES algorithm can be decoded through analyzing sequences of power consumption signals. Michalevsky et al. [58] described an attack that allows a malicious app on Android to learn information about the user's location by reading the phone's aggregated power consumption over a period of a few minutes. By measuring the intervals between keystrokes, Song et al. [59] showed it is possible to recover the password or other sensitive information typed by a victim during SSH sessions. Hund et al. [60] implemented a practical timing side channel attack against ASLR to infer information about the protected address space layout. Andrysco et al. [61] identified a timing channel in the floating point instructions of modern x86 processors, which can be used to break the isolation guarantees of Web browsers.

VIII. Conclusion

In this paper, we describe our finding of a new information leakage channel on Android – interrupt statistical information (`/proc/interrupts`). This channel could leak the running status of devices and be exploited by attackers to infer private information. We propose the interrupt timing analysis as a general approach and demonstrate the practicality with two inference attacks which can infer user's unlock pattern and the app running in the foreground. We implemented attack prototype apps and evaluated them using the real-world data. Our experimental studies show that indeed interrupt statistics could lead to leaks of user's sensitive information or actions. We believe such security threat from the ill-conceived integration of hardware components and tailored kernel is not just an isolated incident and call for the attention from the security community.

Acknowledgements

We thank anonymous reviewers for their insightful comments. This work was partially supported by NSFC (Grant No. 61572415), and the General Research Funds (Project No. CUHK 4055047 and 24207815) established under the University Grant Committee of the Hong Kong Special Administrative Region, China.

References

[1] L. Cai and H. Chen, "TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion," in *Proceedings of the 6th USENIX Workshop on Hot Topics in Security (HotSec)*, 2011.

[2] A. J. Aviv, B. Sapp, M. Blaze, and J. M. Smith, "Practicality of Accelerometer Side Channels on Smartphones," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, 2012.

[3] R. Templeman, Z. Rahman, D. J. Crandall, and A. Kapadia, "PlaceRaider: Virtual Theft in Physical Spaces with Smartphones," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, 2013.

[4] T. Fiebig, J. Krissler, and R. Hänsch, "Security Impact of High Resolution Smartphone Cameras," in *Proceedings of the 8th USENIX Workshop on Offensive Technologies (WOOT)*, 2014.

[5] S. Jana and V. Shmatikov, "Memento: Learning Secrets from Process Footprints," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*, 2012.

[6] K. Zhang and X. Wang, "Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems," in *Proceedings of the 18th USENIX Security Symposium*, 2009.

[7] W. Ogunwale, "Lockdown AM.getRunningAppProcesses API with permission.REAL_GET_TASKS," https://android.googlesource.com/platform/frameworks/base/+/aaa0fee, 2015.

[8] J. Corbet, A. Rubini, and G. Kroah-Hartman, "Interrupt Handling," in *Linux Device Drivers*, 3rd ed. O'Reilly Media, 2005.

[9] D. P. Bovet and M. Cesati, "Interrupts and Exceptions," in *Understanding the Linux Kernel*, 3rd ed. O'Reilly Media, 2005.

[10] P. Brady, "Anatomy & Physiology of an Android," in *Google I/O*, 2008.

[11] "Android Interfaces and Architecture," https://source.android.com/devices/index.html.

[12] "Android Kernel Features," http://elinux.org/Android_Kernel_Features.

[13] J. Levin, "I/O Kit Kernel Drivers," in *Mac OS X and iOS Internals: To the Apple's Core*. John Wiley & Sons, 2012.

[14] M. E. Russinovich, D. A. Solomon, and A. Ionescu, "Trap Dispatching," in *Windows Internals, Part 1*, 6th ed. Pearson Education, 2012.

[15] "Xperf Command-Line Reference," https://msdn.microsoft.com/en-us/library/windows/hardware/hh162920.aspx.

[16] R. Biddle, S. Chiasson, and P. C. van Oorschot, "Graphical Passwords: Learning from the First Twelve Years," *ACM Computing Surveys*, 2012.

[17] S. Uellenbeck, M. Dürmuth, C. Wolf, and T. Holz, "Quantifying the Security of Graphical Passwords: the Case of Android Unlock Patterns," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[18] 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*, 2014.

[19] 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 *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*, 2015.

[20] "SHAREit," https://play.google.com/store/apps/details?id=com.lenovo.anyshare.gps.

[21] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, Location, Disease and More: Inferring Your Secrets from Android Public Resources," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013.

[22] "Capacitive VS Resistive Touch Panels," http://www.newhavendisplay.com/capacitive_vs_resistive.html.

[23] C. Ji, "Internal input event handling in the Linux kernel and the Android userspace," http://seasonofcode.com/posts/internal-input-event-handling-in-the-linux-kernel-and-the-android-userspace.html, 2011.

[24] "Linear Interpolation," https://www.encyclopediaofmath.org/index.php/Linear_interpolation.

[25] "Curve Fitting Toolbox™," http://www.mathworks.com/products/curvefitting/.

[26] L. R. Rabiner and B.-H. Juang, "An Introduction to Hidden Markov Models," *ASSP Magazine, IEEE*, vol. 3, no. 1, pp. 4–16, 1986.

[27] G. D. Forney Jr, "The Viterbi Algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[28] C. C. Aggarwal, "Probabilistic Models for Classification," in *Data Classification: Algorithms and Applications*. CRC Press, 2014.

[29] "XPIN Clip: Bruteforce PIN and PatternLock Solution," http://xpinclip.com/.

[30] A. J. Aviv, K. L. Gibson, E. Mossop, M. Blaze, and J. M. Smith, "Smudge Attacks on Smartphone Touch Screens," in *Proceedings of the 4th USENIX Workshop on Offensive Technologies (WOOT)*, 2010.
[31] "Android NDK," http://developer.android.com/ndk/index.html.
[32] M. Løge, "Tell me who you are and I will tell you your lock pattern," in *DEFCON*, 2015.
[33] A. J. Aviv and D. Fichter, "Understanding Visual Perceptions of Usability and Security of Androids Graphical Password Pattern," in *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, 2014.
[34] C. Haase and R. Guy, "For Butter or Worse - Smoothing Out Performance in Android UIs," in *Google I/O*, 2012.
[35] "Implementing graphics," https://source.android.com/devices/graphics/implement.html.
[36] grafiker, "Display Subsystem," http://droid-grafiker.blogspot.hk/2012/07/display-subsystem.html, 2012.
[37] Android Open Source Project, "DispSync.cpp," https://android.googlesource.com/platform/frameworks/native/+/kitkat-release/services/surfaceflinger/DispSync.cpp.
[38] J. B. Kruskal and M. Liberman, "The Symmetric Time Warping Problem: From Continuous to Discrete," in *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.
[39] S. Salvador and P. Chan, "FastDTW: Toward Accurate Dynamic Time Warping in Linear Time and Space," in *Proceedings of the Third SIGKDD Workshop on Mining Temporal and Sequential Data*, 2004.
[40] "monkeyrunner," http://developer.android.com/tools/help/monkeyrunner_concepts.html.
[41] T. M. Cover and P. E. Hart, "Nearest Neighbor Pattern Classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.
[42] T. Abeel, Y. V. de Peer, and Y. Saeys, "Java-ML: A Machine Learning Library," *Journal of Machine Learning Research*, vol. 10, pp. 931–934, 2009.
[43] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang, "Leave Me Alone: App-Level Protection against Runtime Information Gathering on Android," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*, 2015.
[44] C. Negus, "Managing Disks and Filesystems," in *Linux Bible*, 8th ed. John Wiley & Sons, 2012.
[45] "THE /proc FILESYSTEM," https://www.kernel.org/doc/Documentation/filesystems/proc.txt, 2009.
[46] "irqbalance," http://www.unix.com/man-page/linux/1/irqbalance/.
[47] Q. Xiao, M. K. Reiter, and Y. Zhang, "Mitigating Storage Side Channels Using Statistical Privacy Mechanisms," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.
[48] C. Lin, H. Li, X. Zhou, and X. Wang, "Screenmilker: How to Milk Your Android Screen for Secrets," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.
[49] Z. Qian, Z. M. Mao, and Y. Xie, "Collaborative TCP Sequence Number Inference Attack – How to Crack Sequence Number Under A Second," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, 2012.
[50] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking Kernel Isolation," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.
[51] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang, "ACCessory: Password Inference Using Accelerometers on Smartphones," in *Proceedings of the 2012 Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2012.
[52] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury, "TapPrints: Your Finger Taps Have Fingerprints," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.
[53] Z. Xu, K. Bai, and S. Zhu, "TapLogger: Inferring User Inputs On Smartphone Touchscreens Using On-board Motion Sensors," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2012.
[54] Y. Michalevsky, D. Boneh, and G. Nakibly, "Gyrophone: Recognizing Speech from Gyroscope Signals," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.
[55] H. Wang, T. T. Lai, and R. R. Choudhury, "MoLe: Motion Leaks through Smartwatch Sensors," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2015.
[56] X. Liu, Z. Zhou, W. Diao, Z. Li, and K. Zhang, "When Good Becomes Evil: Keystroke Inference with Smartwatch," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.
[57] P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology - CRYPTO'99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, 1999.
[58] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and G. Nakibly, "PowerSpy: Location Tracking Using Mobile Device Power Analysis," in *Proceedings of the 24th USENIX Security Symposium*, 2015.
[59] D. X. Song, D. Wagner, and X. Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH," in *Proceedings of the 10th USENIX Security Symposium*, 2001.
[60] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks against Kernel Space ASLR," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P)*, 2013.
[61] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On Subnormal Floating Point and Abnormal Timing," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*, 2015.
[62] J. Levin, "The Android Input Architecture," http://newandroidbook.com/files/AndroidInput.pdf, 2015.
[63] "Graphics architecture," https://source.android.com/devices/graphics/architecture.html.

APPENDIX

A. Android Touch Event Processing Flow

Fig. 17 illustrates the touch event processing flow on Android and the implementation of each layer is described below:

- *Hardware Device Layer:* Touchscreen can sense the movement of user's finger on the surface and release an IRQ when detected.
- *Linux Kernel Layer:* CPU responds to the IRQ by calling the handler registered by the input device driver. In this case, touchscreen controller driver decodes the physical signals about the touch action (down / up) to touch location on the screen (Cartesian coordinates) and other useful information. Next, the Linux input event driver – evdev translates device-specific signals into Linux input events and pass them to the character devices (single characters are transmitted) defined in the `/dev/input/eventX` directory [62].
- *Hardware Abstraction Layer:* The EventHub component provided by Android receives the raw input events reported by the kernel and converts them to Android events.
- *Hardware Independent Layers:* Finally, after event decoding (by InputReader) and dispatching (by InputDispacher), the events are delivered to the app taking focus at foreground as MotionEvent objects.

B. Android Display Work Flow

Below, we describe the work flow of DSS through different layers, as shown in Fig. 18. For simplicity, we focus on the flow for 2D frame refreshing [63]:

- *Hardware Independent Layers:* Every window that is created on the Android platform is backed by a Surface, which is used for drawing display content. A Surface could overlap or even override another one and updating

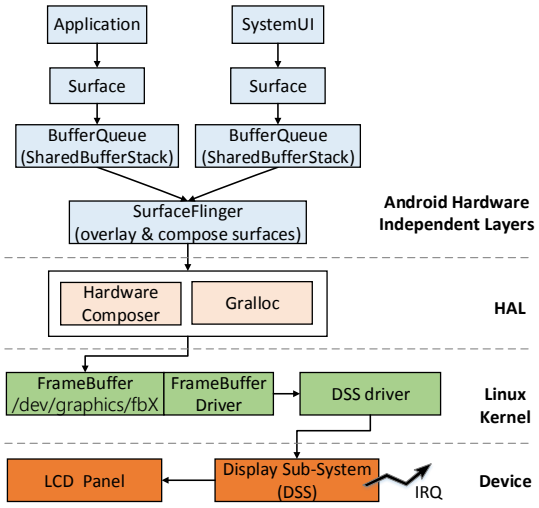

Fig. 18. Simplified Android display system processing flow.

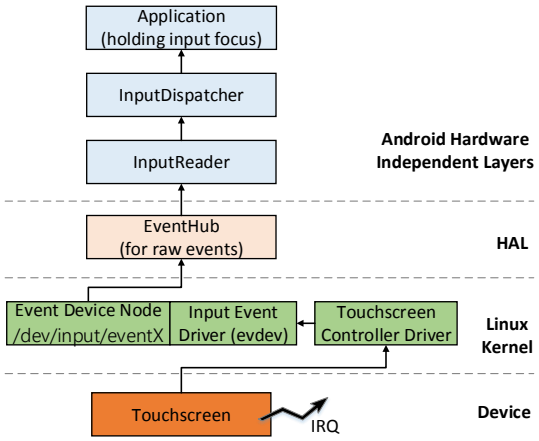

Fig. 17. Simplified Android touch event processing flow.

Surface object would cause the screen refresh. Multiple Surfaces may be active concurrently and they are composited by SurfaceFlinger onto the display.

- *Hardware Abstraction Layer:* Hardware Composer is the central point for all Android graphics rendering, which is used by SurfaceFlinger to composite Surfaces to the screen. The graphics memory allocator Gralloc is responsible for allocating memory that is requested by image producers.
- *Linux Kernel Layer:* Gralloc operates the FrameBuffer in this layer defined as a character device `/dev/graphics/fbX` (`fb0` for the main monitor) with the UI content. Once the FrameBuffer geometry is programmed, the DSS starts pulling the pixels from memory and sending them to the display device [36], which will be refreshed constantly.
- *Hardware Device Layer:* When a screen refresh is completed, an IRQ will be released by DSS.

C. App Activity Detection

Our attack mainly focuses on identifying the launching pattern of an app. This attack could be extended to identify which Activity of a targeted app is running in the foreground through similar methods. With such information at disposal, an attacker can hijack Activities at any time during the lifetime of victim app. The Activities with unique UI refreshing pattern among all the Activities initiated by one app are more likely to be successfully inferred, due to the nature of our attack. The bar for such attack is, however, higher since the loading time of an Activity is usually shorter than the time of app launching.

We find two types of Activities, camera and login, usually have unique UI refreshing patterns and can be reliably inferred. Activity for camera always refreshes its UI for image previewing, so continuous and large amount of interrupts could be observed. A login Activity is often quite different from other Activities as unique third-party modules for SSO (Single Sign On), like Facebook and Google+ SSO modules, are included.

We examined several apps and found indeed these two types of Activities can be attacked. For example, the login Activity of a popular traveling app Expedia can be inferred. We analyzed the Activity transition flow of this app and discovered that the main Activity leads to any one of 9 Activities based on user's choice. Among them, some Activities are indistinguishable, like `PreferenceActivity` and `AboutActivity`, because their UIs are very concise and contain no dynamic data. However, `AccountLibActivity` (for login) is quite different from other 8 Activities. We collected the interrupt time series for all 9 Activities and applied the same training and testing methodologies. It turns out `AccountLibActivity` could be detected by our method with 100 % success rate. Therefore, an attacker targeting the credentials of Expedia users can achieve her goal leveraging the information provided by our inference attack.

Previous work on shared-memory side-channel by Chen et al. [18] studied the problem of UI hijacking on Android as well. Their work combines other data sources like CPU utilization time, network statistics to reach high accuracy. Our initial result shows interrupt patterns itself could be used to construct fingerprint. We believe our attack would also benefit from using these sources as well.

D. Experimental Dataset for Unlock Pattern Inference Attack

2-gram Patterns.

```
1852    2584    2586    2963    3216
3576    3692    4951    5147    6547
6741    7456    78951   7896    8524
8529    8753    95147   9635    98741
```

3-gram Patterns.

```
123456  29516   36947   3854    4153
4983    5214    5491    5693    5729
65481   6745    6849    7234    7486
7594    81476   8549    9213    951234
```

4-gram Patterns.

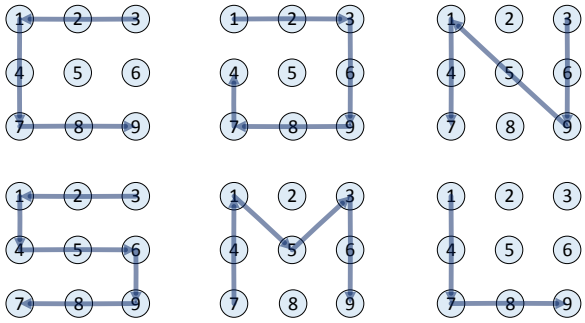

Fig. 19. Popular patterns.

```
124567  18579   275389  278945  29587
35918   389514  51897   52146   549637
635742  63894   6741258 743218  76941
78365   841596  87253   94571   78963214
```

5-gram Patterns.

```
1258469 1485263 1598436 186347  2586793
3269514 3572814 418365  451823  36214789
451863  452871  4571238 5283147 543689
576483  6753291 749568  7534921 74168523
```

Popular Patterns.

```
1235789 1235987 1236987 12369   1475963
14789   1478963 3214789 3215987 36987
123654789  12369874  147852369  14789632
321456987  36987412  7415369    7415963
```

E. Training Dataset for Foreground App Inference Attack

```
air.com.hoimi.MathxMath
cmb.pb
cn.etouch.ecalendar.longshi2
cn.wps.moffice
com.aastocks.dzh
com.airbnb.android
com.ajnsnewmedia.kitchenstories
com.android.phone
com.antivirus
com.antutu.ABenchMark
com.baidu.baidutranslate
com.baidu.baike
com.baidu.searchbox
com.booking
com.citrix.saas.gotowebinar
com.cleanmaster.mguard
com.cleanmaster.security
com.coolmobilesolution.fastscannerfree
com.csst.ecdict
com.dewmobile.kuaiya.play
com.dianping.v1
com.douban.frodo
com.eastmoney.android.fund
com.exchange.rate
com.facebook.pages.app
com.facebook.work
com.globalsources.globalsources
com.google.android.apps.docs
com.google.android.apps.docs.editors.docs
com.google.android.apps.docs.editors.sheets
com.google.android.apps.docs.editors.slides
com.google.android.deskclock
com.google.android.googlequicksearchbox
com.google.android.keep
com.google.android.street
com.hket.android.ctjobs
com.hse28.hse28
com.htsu.hsbcpersonalbanking
com.icoolme.android.weather
com.imdb.mobile
com.indeed.android.jobsearch
com.intsig.BCRLite
com.intsig.camscanner
com.job.android
com.jobmarket.android
com.jobsdb
com.Kingdee.Express
com.kpmoney.android
com.labour.ies
com.lenovo.anyshare.gps
com.linkedin.android.jobs.jobseeker
com.lionmobi.battery
com.lionmobi.powerclean
com.magisto
com.malangstudio.alarmmon
com.mandongkeji.comiclover.play
com.megahub.appledaily.stockking.activity
com.microsoft.hyperlapsemobile
com.microsoft.rdc.android
com.miniclip.agar.io
com.mmg.theoverlander
com.mobilesoft.kmb.mobile
com.mobisystems.office
com.money.on
com.mtel.androidbea
com.mt.mtxx.mtxx
com.mymoney
com.nuthon.centaline
com.openrice.android
com.pps.app
com.qihoo.security
com.roidapp.photogrid
com.sankuai.movie
com.scb.breezebanking.hk
com.scmp.jiujik
com.Slack
com.smartwho.SmartAllCurrencyConverter
com.smule.singandroid
com.sometimeswefly.littlealchemy
com.surpax.ledflashlight.panel
com.ted.android
com.tripadvisor.tripadvisor
com.twitter.android
com.wacai365
com.xunlei.downloadprovider
com.yahoo.infohub
com.yahoo.mobile.client.android.weather
com.yipiao
com.youdao.dict
com.zhihu.android
ctrip.android.view
freelance.flk.com.myapplication
io.appsoluteright.hkexChecker
io.silvrr.silvrrwallet.hk
jp.united.app.kanahei.money
me.chunyu.ChunyuDoctor
sina.mobile.tianqitong
tools.bmirechner
tv.danmaku.bili
tw.com.off.hkradio
```